

Kenji Elzerman

From SQL to C# code

Entity Framework Core

Start with Entity Framework Core and implement
it in your own applications in no time!



KENS LEARNING CURVE
I LOVE TO LEARN

Hello there!

Running software isn't all about code, some buttons, and some user interaction. There is much more to it. One of the key elements is data. Data is always data; information a user can see and manage. But also something you, the developer, can work with.

But the way we can save that data is a whole different story. Where do you save information? How do you retrieve it again? How do you make sure the data is saved? How can my mobile app reach data all over the world? For this, we use (SQL) databases.

While Entity Framework is really popular these days I believe it is important to know and understand the basics of the underlying system of Entity Framework Core. That is why we explore SQL before heading to the main guest: Entity Framework Core.

This book is for those who want to learn how to use Entity Framework from zero or those who want to learn a little bit extra about this subject. Don't expect 100% coverage on the whole Entity Framework Core subject, because that's impossible. I will give you enough information to work with in a way you can implement Entity Framework in your own application(s).

Maybe I do things a little bit differently. Most books I have read are so focused on theory and working with the perfect scenario. In my opinion, this isn't right. There isn't a perfect scenario in a perfect environment. That is why I place my focus more on the practical side of things. I want to show you how it is done, rather than just tell you. This way I will tell you less unused theory, but more practical examples.

To help you with the practical side, I have included examples and a project which you can download from my GitHub. With this project, you can type along as we are building more aspects of Entity Framework Core in a console application and API.

This could mean that some subjects are not 100% clear or are marked as nice to have. Don't be afraid to do some investigation on your own. Use Google, DuckDuckGo, Bing, or whatever to help you. In the end, a developer's time is 75% searching on the internet. Do know that you need to know the basics of C#! Know what variables are, how to create classes and methods, how dependency injection works, how to

make decisions, how unit testing works, etc.

Table of Contents

Start

Hello there!

Let me introduce myself

Chapter 1: Before we start

Practice makes perfect

Chapter 2: It all starts with a (SQL) database

In this chapter

Let me introduce you to the database

The SQL Server Object Explorer

A database and a table

A new database

Add a table

Auto-increment

The primary key

Data types

Saving the table

Changing the table

View and manage data

Queries

Select

Sorting

Searching

Insert

Update

Delete

Assignment

Joins

Inner join

Left join

Right join

Full join

Aliases

Column aliases

Table aliases

Transactions

Rollback

Commit

If-Else

Stored procedures

Final words

Chapter 3: Type-Along project

GitHub

Projects

MovieLibrary.Business

MovieLibrary.Domain

MovieLibrary.API

MovieLibrary.ConsoleApp

Chapter 4: Entity Framework Core

In this chapter

Introducing Entity Framework Core

Entities

Database first vs code first

Packages

Context

DbContext

The class

Connection string

Adding the connection string

DbSet

Migrations

- Creating a migration
- Snapshots
- Migrations
- Updating the database

Refactoring the code

- Initializing the DataContext class
- The Get() method
- The Get(int id) method
- The Get(string query) method
- The Create() method
- The Delete() method
- The Update() method

Final words

Chapter 5: Setting up entities

- In this chapter
- Contents of this chapter
- Data annotations

- MaxLength
- Key
- Column
- Required
- DatabaseGenerated
- NotMapped

Assignment

- Connecting the actor
- An actors-class

Act on states

- The ChangeTracker
- Better approach: Interfaces

Final words

Chapter 6: Manage Migrations

In this chapter

Contents of this chapter

Reverting Migrations

- Update database to the past

- Remove the migration file

Adding extra information

- Adding SQL

Adding migrations

- OutputDir

- Stored procedures

Seeding

- Overriding OnModelCreating

- HasData

- The migration

Extras

- Scripting migrations

- Starting over

- List migrations

Assignment

- Adding seeding

- Seeding actors

Final words

Chapter 7: Relationships

In this chapter

Theory

Conventions

Navigations

Reference navigations

Collection navigations

Configuring navigations

One-to-one relation

The Genre entity

Connecting the entities

Related data query

Eager loading

Lazy loading

Explicit Loading

Which one to use?

One-to-many relation

ICollection

Retrieving the data

Join tables

Entity for the join table

Add navigation property
to Movie

Including

Final words

Chapter 8: Stored procedures

In this chapter

A simple stored procedure

An entity

Execute the stored procedure

The ModelBuild

Stored procedure with parameters

Adding to C# and Entity Framework Core

Final words

Chapter 9: API Implementation

In this chapter
Why use dependency injection?
Storing the connection string

A new section
Retrieving the connection string

Adding DataContext to services

AddDbContext
A DataContext constructor

Using the injection

Injecting into Movies

Migrations

No database provider
Missing package

Fixing the console app

Getting the configuration
Setting the environment

Final words

Chapter 10: Logging

In this chapter
The simple way

Console logging
Debug logging
Sensitive data

Log levels

Microsoft logging

Creating LoggerFactory

Sensitive data and log levels

Final words

Chapter 11: Inspecting the queries

In this chapter

ToQueryString

Query interceptors

Create the class

Using the interceptor

Modify the query

Preparations

Expanding the
interceptor

Logging the result

Create the override

Performance monitoring

Monitor the other
overrides

Final words on
performance monitoring

Security

Environment specific

Final words

Chapter 12: Tracking

In this chapter

- Tracking entities
- Attaching entities
- No tracking
- Identity resolution
- Tracking stored procedures
- Final words

Chapter 13: Fluent API

- In this chapter
- Setting up entities

- Required
 - NotMapped
 - Indexes

- Shadow properties

- Creating the property
 - Setting a value

- Final words

Chapter 14: Unit testing

- In this chapter
- Setting up the test project
- In-memory

- Setting up the database
 - A first test
 - Adding a movie
 - Creating the Id for in-memory only
 - Reset the in-memory database

- Final words

Appendix I: To SaveChanges or not to SaveChanges

- To the test!
- Be careful though

AI: Encrypting data

Value converter

Encrypting and decrypting

Some cons

Conclusion

Entity Framework Core

PUBLISHED BY **Kenji Elzerman**

Copyright © 2024 by Kenji Elzerman All Rights Reserved. FIRST
EDITION: July 2024

Cover and interior layout design by Kens Learning Curve / Kenji
Elzerman.

Copyright

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means including information storage and retrieval systems, without written consent from the author. The only exception is by a reviewer, who may quote short excerpts in a review.

This ebook is licensed for your personal enjoyment only. This eBook may not be re-sold or given away to other people. If you would like to share this book with another person, please purchase an additional copy for each recipient. If you're reading this book and did not purchase it, or it was not purchased for your use only, then please purchase your own copy. Thank you for respecting the hard work of this author. You may print out one copy for your personal reference.

Trademarks

This book is in no way affiliated, authorized, sponsored, or endorsed by Microsoft, Entity Framework Core, or any other companies mentioned in this book. All references to Microsoft and other trademarked properties are used in accordance with the Nominative Fair Use Doctrine and are not meant to imply that this book is a Microsoft product for advertising or other commercial purposes.

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. The author cannot attest to the accuracy of this information. Use of a term in the book should not be regarded as affecting the validity of any trademark or service mark.

Let me introduce myself

I have been a developer since 2002 and I have been in many companies and learned a lot (mostly how not to do things).

Then in 2016, I went on a world trip, traveling 1 1/2 years. In this period I learned so much and the feeling of freedom was overwhelming. But as usual; all good things have to end sometime and I had to return to my home country. In 2018 I was back and did what I always did: Find a job, sit in a team, and sit in silence while I developed software.

In 2021 I changed my profession from developer to IT teacher, mostly based on C# development. I started with a small group of juniors being readied for the big bad work called 'working at a company'. After that I started as a teacher for people with ADHD and autism; one of the best and most rewarding jobs I ever had.

But the traveling was calling again and I started my own website; kenslearningcurve.com where I help people learn and grow in the world of programming. And I love every minute of it.

Now I want to share my knowledge and experiences with everyone around the world.

Welcome to my e-book “Entity Framework Core”. My first e-book where I will tell you almost everything I know about Entity Framework Core and a little bit extra.

Chapter 1: Before we start

In this chapter, I will explain to you (the reader) what you can expect and how this e-book works. Most people will skip this chapter and dive right in, but I want to invite you to stick around and read what my idea of a book - and learning - is all about.

Practice makes perfect

While most books will tell you everything you should know in perfect scenarios and leave you with a lot of information, I want to challenge you to do as you learn.

That is why I created a start solution with some basic projects and code. This solution is the base of our study and learning. In each chapter, I will give you examples and you can implement those examples in the start solution.

A few chapters end with assignments. You need to do them before you can continue; the next chapter will use those assignments as examples or to continue building on the projects.

At the end of this e-book, you should have a complete application with all functionality using Entity Framework Core to start with. You learned a lot of terms you can use to explore more of this magnificent ORM.

This way you can not only read about something but actually experience it. A way of learning that I think works better and keeps your attention to the information.

Chapter 2: It all starts with a (SQL) database

This chapter is a little bit different than the others. I am not going to talk about C# or Entity Framework Core, but databases and SQL. Some important information if you want to work with databases.

If you know (T)SQL and you think you know the terms query, join, and transactions and know how to create a database in Visual Studio, you can skip this chapter and move to the next. But I will be using a database in the next article that I am making in this one. It's up to you.

In this chapter

In this chapter, you will learn the **basics of a database**. This isn't about Entity Framework Core yet, but some aspects of SQL would help you understand Entity Framework Core better.

We will look at how you can **create a database** and how to **manage the data inside** the database. To store the data inside a database you will need a table. We will also be looking at **primary keys**, **different data types**, and **auto-increment**.

Another big aspect of this chapter is to learn how to **create and execute queries**. You can retrieve, create, delete, or edit the data inside a database table with queries.

When we manage the basics of creating and executing queries, we will take it up a nudge and combine different data by using **joins** and combining data from different tables. Joins might cause some ambiguous names, so we will dive right into the **aliases** after the joins.

To avoid problems and remove or edit data by accident we use **transactions**. We can “test” a query before committing it.

At the end of this chapter, we will take a quick look at **conditions** inside SQL and **stored procedures**.

We will use the **SQL Object Explorer** inside Visual Studio to manage

all this. A simple, yet powerful, window inside the IDE that does everything we need.

Let me introduce you to the database

Before diving into C# or Entity Framework Core and getting data from a database or sending information back, let's look at what a database is and how it all generally works.

There are four well-known and well-used database systems: MySQL, MSSQL, Oracle, and PostgreSQL. If you know how to work with one it isn't hard to learn to work with the other. Remember that I will be using MSSQL (Microsoft SQL) in my examples.

A database is a structured collection of data. It is organized, managed, and stored in a way we can easily retrieve, change, and send data. They help us centralize data so it can be accessed by different clients. Clients as in native applications, web applications, mobile applications, etc.

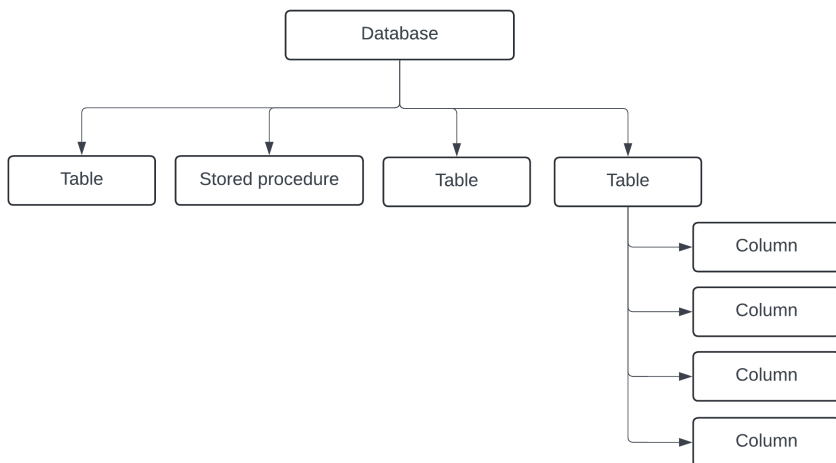
A database has a schema or objects. The most important one is tables. A table represents the data by defining columns. Each column has a certain type, just like a C# variable. Those types look like C# data types but are a bit different. The type of the column defines what kind of data can be placed in a cell. A cell is the row and column of the table.

A column can only have one data type. Changing this could cause data loss or errors (hope for the second one if you change it by accident). Here is a representation of how a table inside a database could look:

	TITLE	RELEASE DATE	SEEN	RATING
ROW 1	The Matrix	199-03-31	1	8
ROW 2	Shrek	2001-04-22	0	9
ROW 3	Jaws	1975-06-20	1	7
ROW 4	Inception	2010-07-22	1	5

A table has a name and so do the columns. Here the same rule goes as with C#: Keep them short and simple, but they should tell what the table or column is about. A database can contain more than one table and each table can have several columns and rows.

Another important object to know is a stored procedure. Later you will learn how you can collect and manage data with queries. A stored procedure can be explained as a method in C#; It can do more than one thing with multiple lines of code. The lines of code in a stored procedure are not C#, but it's TSQL. Stored procedures are used to execute complex queries and manage data on a high level and the server.



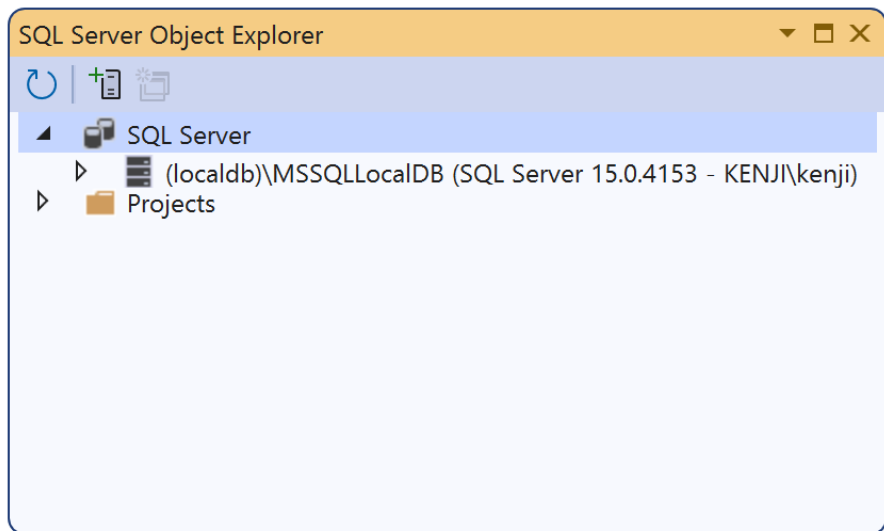
If you have Visual Studio installed you have a localdb which is a local, simplified version of MSSQL. I will be using this throughout this article. But it is possible to install and/or use a SQL server.

The SQL Server Object Explorer

Had enough theory? Well, I do and I want to dive into Visual Studio. There are many tools to manage a database, which you need to do at some point. In the past, we used SQL Management Studio, but I find it a big tool for the small things I am doing with a database.

Lucky for us Visual Studio comes with some sort of database management window: **The SQL Server Object Explorer**. A nice little window with a lot of options. Let's start by making it appear in your Visual Studio, as you might not have it... Yet.

To open the SQL Server Object Explorer, open the View menu in Visual Studio and select SQL Server Object Explorer. Or press CTRL + Q, type 'sql server', and double click SQL Server Object Explorer.



If you expand the SQL Server, you will see something like (localdb)\MSSQLLocalDB. This is your local MSSQL database you can use for development purposes. It is here where we will create and manage the databases.

Expanding this local database you will see three folders:

Databases

Here you will find your databases when you create them. The System Databases keeps track of all objects on the server. Whatever you do, never delete this one!

Security

Handles everything that is in some way connected to the security of your databases and database server (the localdb).

Server Objects

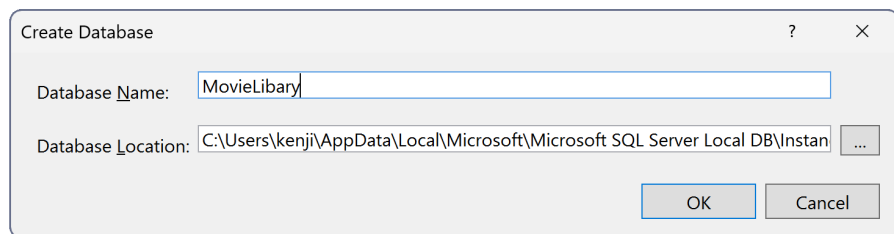
This folder contains objects associated with the SQL Server instance, not tied to specific databases. Think of backup devices, linked servers, server triggers, endpoints, and audits. They are crucial for administration, security, and network communication at the server level. We will not touch these in this book.

The one you will use the most is **Databases**.

A database and a table

A new database

Right-click on that folder Databases and select Add New Database. Still with me? Good! Now type a name. If you want to follow this book to the letter, use the same name as me: MovieLibrary. I keep the default database location but feel free to place your database somewhere else.



The database should be created and placed underneath the folder System Databases. If you expand your new database, you will see 8 new folders. You won't need all of them, so I will only explain the important ones:

Tables

Here you will find your tables and the data inside the tables. We will come back to this later.

Views

A view is a combination of multiple tables, generating a new table for you to use. This is read-only and you can access it like a table.

Programmability This one has more folders. But here you will store your stored procedures and functions. Functions are somewhat the same as stored procedures.

Security

The same as the Security folder of the database server, but this one only manages the security of the database. It can override the settings of the database server security.

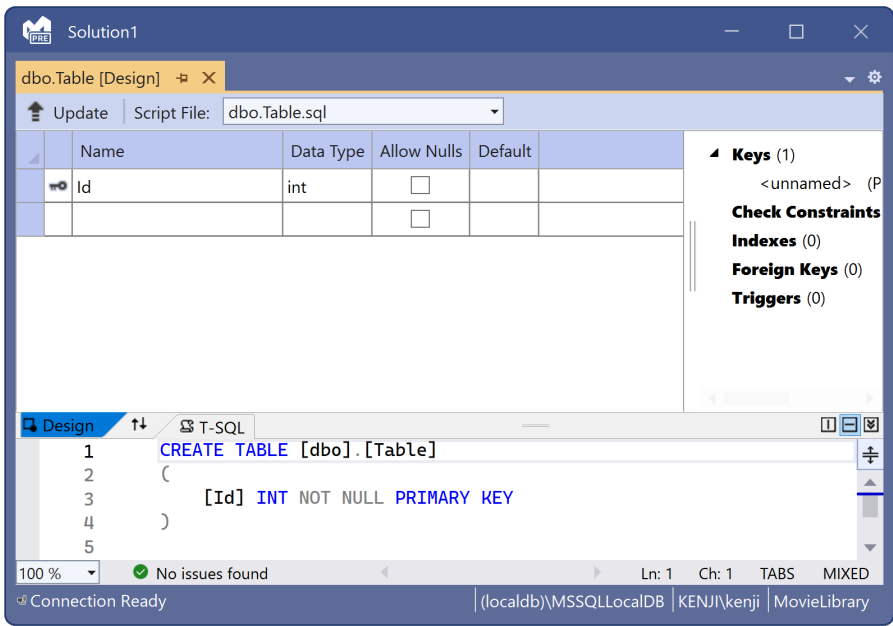
While you won't use most of them, know it is there and you could use it if you need it. Let's create a table.

Add a table

If you expand the folder *Tables*, you will see three folders. I will not go into detail since all the minor functionalities of a database server,

database, or table will take too much of our time.

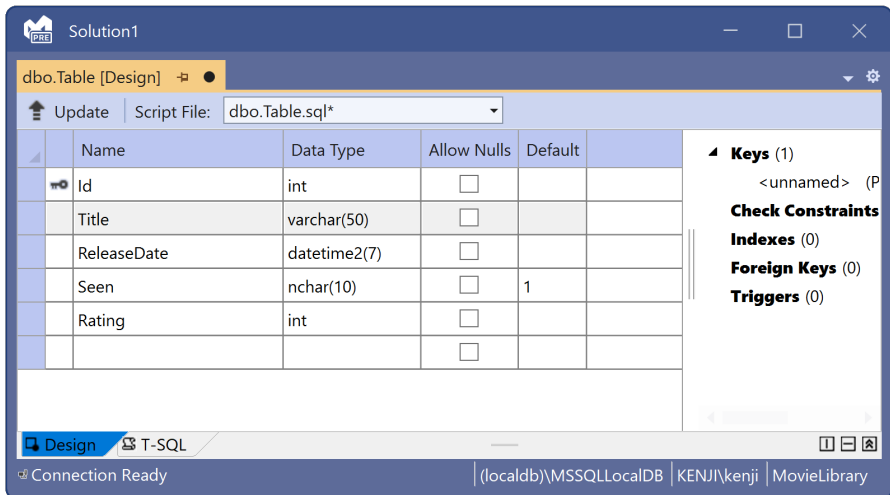
Instead, we are going to create a new table. This works almost the same as creating a database: Right-click on the folder *Tables* and select *Add New Table*. A somewhat scary editor is opening. It is here where we will define our columns, including the data types of the data.



The above part is how you can create your columns or fields. Ironically enough you create a table in a table. You need to give each field a name, and a data type, and allow NULL as a value (yes/no), and a default value. Name and data type are required, others are optional (Allow Nulls is a boolean, so it's false by default).

The lower part represents the table and the fields you create in the upper part. This is TSQL; you can copy and execute it later.

Let's create our table! I want to store movies here, as I showed you in the image in the chapter *Let me introduce you to the database*. Fill in the grid/table as follows:



Auto-increment

The Id is still there and for a good reason. Each row in a table can have a unique identifier. This way we can retrieve just that one row, or record, that we want. An ID is usually an int, but it could also be a long (or a bigint in SQL) or a GUID. An ID should be unique and auto-increment, meaning that the value of the ID is always upped when a new record is added automatically. You don't have to set the ID by hand. You won't be able to add or edit the value.

But the auto-increment isn't set by default. If you select the ID and press F4 (or open the properties some other way) you see all the properties for this column. First, you see the general stuff we already set. But if you go lower you will find the **Identity Specification**. Expand this one. Now set (Is Identity) to true. The Identity Increment and the Identity Seed are both set to 1.

The Identity Increment is the number that will be used to add to the maximum value (the latest added ID). The Seed is the initial value of new rows. In this case, a new row will set the id to 1. But if we add a new row, the ID of the new row will be 2 (1 of the last row plus the number of the increment value).

The primary key

If you look carefully at the ID you will notice the little key on the left. This means the ID is a primary key. A column that is marked with a primary key can only contain unique values. Which is convenient for

an ID, since we don't want duplicated values for a unique identifier.

Another benefit of a primary key is that the data is indexed, meaning that it will improve the performance for faster look-up and retrieval of the data based on that field. Usually marked as a primary key PK, also helps create relationships between other tables. Because the PK is unique, you can connect other rows from other tables to one row in another table. More about this later. Each table can only have one primary key. If you want similar functionality to a column you can create a *composite primary key*. But don't bother about that too much for now.

Data types

As you can see the ID has a data type of int, which has the same rules/benefits/properties as an int in C#. The title however has a data type of varchar(50). This one is similar to a string. The 50 means that it only allows up to 50 characters in the value. More will be ignored or you get an error (depending on what you use to talk to the database).

The **ReleaseDate** is of the data type datetime2(7). There are other types for date and time, but this one suits the best for normal dates and times.

And then there is a bit. A bit can be compared to a boolean. A bit is 1 or 0 (true or false). Pretty simple, right?

checked the *Allows Null* for **Rating** because I don't always want to enter a rating. Especially when I haven't seen the movie yet, when this one is not checked, the default value will be required. If you don't set a value when inserting data, you will get an error.

Saving the table

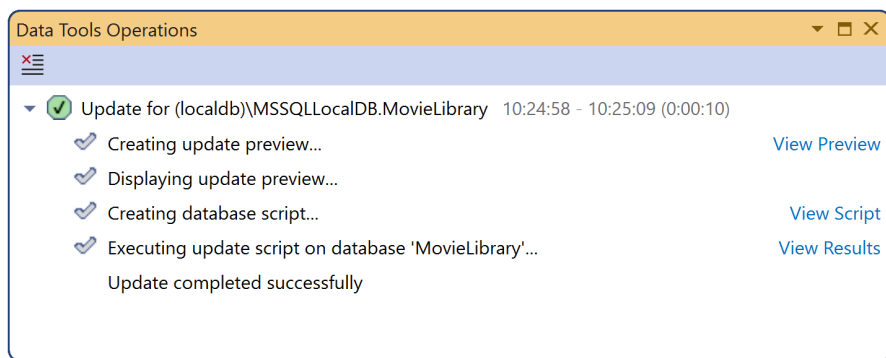
Before we can create the table; have you noticed we didn't add a name for the table? This is a bit of an annoying part because the name of the table needs to be set in the TSQL (the lower part).

Changing it by changing the SQL means you need to replace the [Table] with [YourName]. In this case, it should be [Movies]. Although you will find the name in the properties of the table, you can't change it there. You need to change the name of the table in the SQL.

Awesome! Now you are all set to update the database. In the top left corner of the editor is an up-arrow with the word *Update*. Press that one. A window appears that will do some magic you won't see and eventually show you a preview/summary. You have three options:

- **Generate scripts**
Choose this if you want to generate an SQL file that you can execute at any time.
- **Update Database**
This will generate the script in memory and execute it against the database.
- **Cancel**
Closes the window and does nothing.

Choose the second option. Give it a few seconds and your table will be created, as you can see in the output.



Changing the table

Maybe you made a typo - happens to the best – or you want to add or remove a column. You can easily fix this by opening the table designer again. You do this by right-clicking on the table in the SQL Server Object Explorer and selecting **View Designer**. This will open the designer, which is the same window you use when creating the table.

Let's add a plot for the movie. The plot is usually a short description of the movie, but could also be pretty long. Anyway, it's more than 50 characters. It's more a text.

We add a column by just adding it like when we created the table. Name the column Plot and choose the data type nvarchar(500). You can change the number to what you would like. There is nvarchar(MAX), which means that you can store data up to 2GB. Don't

just select it, but consider thinking about it twice before you use it.

Uncheck the Allow Nulls and no default value. Press the Update button and let Visual Studio update your database.

Close the designer, we will no longer need it.

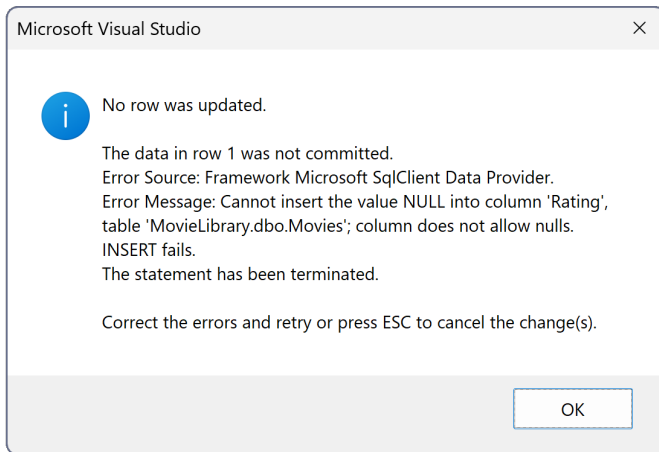
View and manage data

Now that we have a database and a table, it would sound like a good idea to add some data. Why else would you need a database?

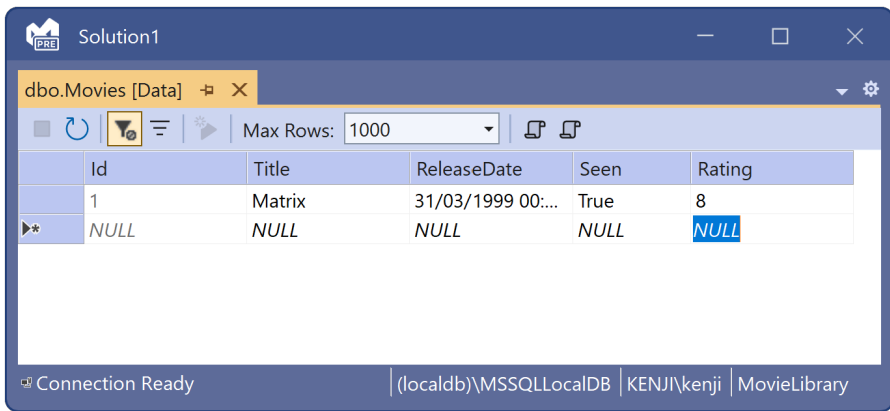
Right-click on the table and select **View Data**. You now see the table with the fields and cells under the headers that all say NULL. This way you can view and manage the data in the table.

This row with the NULLs is a new row you can add. All you now have to do is fill the cells, except the ID. This one is auto increment, remember? And you can't even change the value, it's disabled.

If you make a typo or you don't enter data in a column that doesn't allow nulls, you will get an error like this:



Fix the error and press enter. The row will now be added:



Now, continue adding the other movies as shown in the image from the chapter *Let me introduce you to the database*.

Let's remove the movie Inception. It wasn't a good movie anyway (PERSONAL OPINION!). You can easily remove a row by right-clicking anywhere on the row and selecting **Delete**. Then confirm you really want to delete that row. And POOF! Gone. And gone forever, so be careful with this.

To change a value in a cell, simply change the content and press enter. Let's change the movie Shrek to Shrek The First and change the rating for Jaws from 7 to 8.

Let's add the movie The Muppets Take Manhattan (July 13, 1984). Add your own rating, plot, and whether you have seen it or not.

Notice that the ID for The Muppets isn't 6 (maybe in your case 5, since mine starts at 2... Don't ask). The movie Inception was 6, but we deleted that one. So why isn't it 6? SQL always remembers the last ID and adds the increment value to it. So Inception was 6, the highest ID at the time was 6, Inception is deleted and the highest ID is still 6. Adding a new record would get the ID with a value of 7.

Enough Visual Studio and the SQL Object Explorer. Let's look at awesome queries!

Queries

Alright, something different; queries. Maybe you have seen them, perhaps you haven't. I rarely use them these days. It's mostly because Entity Framework Core will remove the need to write queries, but it's

still a good idea to know what queries are and how they generally work.

Queries are written requests to retrieve, manage, and analyze data that is stored in the database. We write queries with SQL, which stands for **Structured Query Language**. It feels and looks a bit like English combined with search parameters.

There are different scenarios that you can achieve with queries. You might want to retrieve a specific set of data from a table, delete certain records, update records, or do other expert stuff.

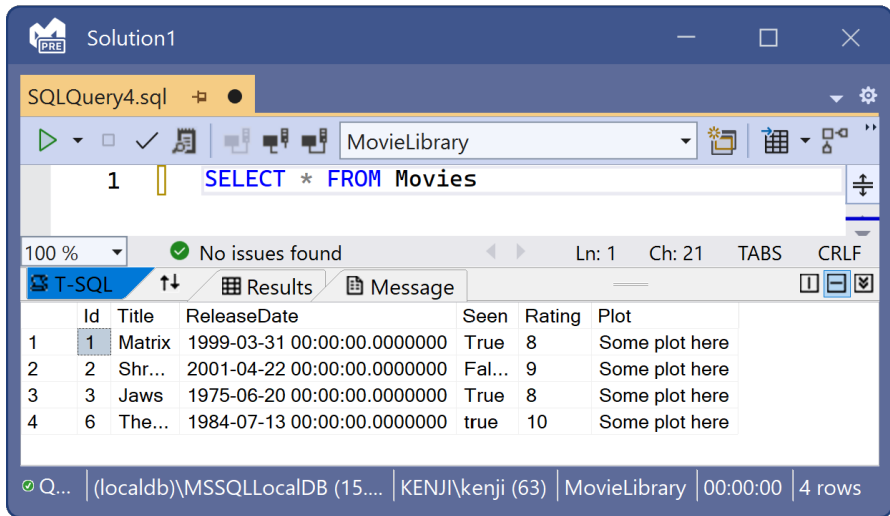
To write and execute a query you need a query editor. Lucky for us this is something which is inside Visual Studio. Right-click on the database MovieLibrary and select **New Query**. A new editor window is opening, just like with a C# file. But this time it is a SQL file.

Type the following SQL statement in the editor:

```
SELECT * FROM Movies
```

Press the green play arrow inside the editor, not the run button you usually use. This is a bit confusing at first. The green play button in Visual Studio, the top one, runs the application. The one inside the query editor runs the query. Also, don't press F5, this will also run the application and not the query. If you want to use the keys, press CTRL + SHIFT + F5.

You will now see a grid with the result of the query. How this query works will be explained later. Just keep in mind that SELECT means you want to retrieve something from a table.



And this is how you can execute queries. But how do they work? There is something called CRUD, which stands for **C**reate, **R**ead, **U**pdate, and **D**eleate. And queries can do just that. By giving a command, or several commands combined, you can do just that.

You can expand a query to pinpoint specific records you want to retrieve, update, or delete. Or you can add specific data to a table. Let's look at the different kinds of queries.

Select

The select query is specifically designed for retrieving data from a database, or a table to be more exact. The select is the read of the CRUD and the easiest to use and understand.

A simple select command looks like this:

```
SELECT * FROM Movies
```

The wildcard means all columns, so if we translate this query to basic English: select the data with all columns from the table Movies. This results in all rows currently in the table Movies.

But imagine you have a table with 100.000 movies. This could be a performance issue if you retrieve all records in the table. Especially when you only want a specific set, let's say the movies you haven't seen yet (seen = false). Then we can use the where clause. The where allows you to filter the selection. For example:

```
SELECT * FROM Movies WHERE seen = 0
```

If you follow this book from the start and execute this query you will get one record: Shrek. This is correct because it is the only movie I haven't seen yet. You can look at the WHERE as the start of and the conditions behind it. You can combine where clauses. Here is an example:

```
SELECT * FROM Movies WHERE rating > 7 AND title LIKE '%hr%'
```

This will again show Shrek because this movie has a rating higher than 7 and the title contains 'hr'. Don't bother about the LIKE, that will be explained later.

If you want to select specific columns you can replace the wildcard with the names of the columns. We usually do this to increase the performance a little bit by getting rid of information we don't want.

```
SELECT title, seen, rating FROM Movies
WHERE rating > 7
AND title LIKE '%hr%'
```

Sorting

We will continue with the SELECT, but add a sorting. Sorting means that we can sort the data on a column from A-Z or Z-A (0-9 or 9-0). We sort data by using the ORDER BY clause. This clause is placed after the WHERE clause and it looks like this:

```
SELECT title, seen, rating FROM Movies
WHERE rating > 7
ORDER BY title
```

The default sorting is always ascending, meaning from A to Z (or 0 to 9). If you want to turn it around you have to use the descending ordering, which is DESC in SQL and placed after the column you want to sort on:

```
SELECT title, seen, rating FROM Movies
WHERE rating > 7
ORDER BY title DESC
```

If you want to order by ascending, but don't trust the default ORDER BY, you can replace the DESC with ASC and your ORDER BY will order the data ascending.

But we can sort in C# code, right? So why sort in SQL when we can

do it in code?"

Good question and good point, but keep in mind that sorting in C# costs more performance and memory than sorting it using SQL. Again, imagine you have 100.000 movies and you select them all and retrieve them in C#. Then you need to sort 100.000 movies in C#, which costs more performance and memory than letting the SQL server handle it.

Searching

Searching in SQL is a combination of the WHERE clause and the SELECT. But we can also search for the contents of the cells in a table. We do this with the operator LIKE, which you have seen already. The LIKE operator allows you to only select those rows where a specific cell contains a specific value.

If you use the statement below you will get a single record that results in the movie Shrek:

```
SELECT title, seen, rating FROM Movies WHERE title LIKE 'shrek'
```

But if you use 'sh' instead of 'Shrek', no results will be shown. This is because this notation means that the contents of the title need to be 'sh', and we don't have that. But if you use the notation 'sh%' you will get one record; the movie Shrek.

The % represents any sequence of zero or more characters. Placing it at the end of the string (sh) means that the records with titles that start with 'sh' (Shadowhunters, Shrek, Shazam!, etc) are shown:

```
SELECT title, seen, rating FROM Movies WHERE title LIKE 'sh'
```

Placing the % at the beginning only records with the titles that end with 'sh' (Flash, Rush, Wish) will be shown:

```
SELECT title, seen, rating FROM Movies WHERE title LIKE '%sh'
```

But placing the % at the beginning and end will get all the movies that have 'sh' somewhere in the title (The Shawshank Redemption, Crash, Shrek).

```
SELECT title, seen, rating FROM Movies WHERE title LIKE '%sh%'
```

You can still add more WHERE clauses and add an ORDER BY if you want.

Besides the % operator, there is also an _ operator (underscore). This one does the same as the %, but where the % means zero or more characters, the _ means only one character. See the difference below:

```
-- This will show Shrek, since rek fits in Shrek
SELECT title, seen, rating FROM Movies WHERE title LIKE '%rek'

-- This will show nothing since no movie title has one character
and then rek (Shrek is two characters before rek)
SELECT title, seen, rating FROM Movies WHERE title LIKE '_rek'

-- Will show one result, because the 'S' is the only character
before hrek.
SELECT title, seen, rating FROM Movies WHERE title LIKE '_hrek'

-- Will show one result, since it doesn't matter how many
characters are before hrek.
SELECT title, seen, rating FROM Movies WHERE title LIKE '%hrek'
```

Side note: Yes, you can execute more than one SQL statement in one query. Just put them all in the editor and press run. Each SQL statement will result in a single result. The statements above will produce 4 grids. **Side note 2:** The —(double dash) behind each statement is a SQL comment. In C# we use the //, but SQL uses the –.

You can combine more than one WHERE clause. Simply join them with an AND or an OR. AND means that both clauses should be true and OR means that one or more can be true. For example, I want to retrieve movies where the title contains an 'x' or a 'k':

```
SELECT title, seen, rating FROM Movies
WHERE title LIKE '%x%' OR title LIKE '%k%'
```

This should result in 3 records for the movies The Matrix, Shrek, and The Muppets Take Manhattan. Now replace the OR with AND. No results are shown because there isn't a movie with a title that contains an X and a K.

Insert

In an upcoming chapter, we will implement Entity Framework Core into a console application. One of the things we will do is send data to the database. This will be done through an insert query.

An insert query has a few parts: The INSERT, the table you want to insert the data into, the fields you want to add data in, and the data. If

you want to write an insert query for the Movies table this is how it looks:

```
INSERT INTO Movies (title, releaseDate, rating, plot, seen) VALUES  
( 'Inception', '2010-07-22', 5, 'Still don''t get it', 1)
```

INSERT INTO means that you want to start an insert statement. Then you note the table, Movies in this case. Next are the fields or columns you want to use and have data for. As you might have noticed, I didn't stick to the same order as in the table. As long as you stick to the same order, this is not a problem.

Then VALUES indicates you are giving the values for the fields. For the plot you see don't with two single quotes. This is to escape the single quote because it is used to indicate strings. A number (int) doesn't need quotes.

Again, you don't fill in the ID; this one is generated by the database. Adding it yourself will only result in errors.

If you add a select directly after the insert you will see the newly added record in the table.

```
INSERT INTO Movies (title, releaseDate, rating, plot, seen) VALUES  
( 'Inception', '2010-07-22', 5, 'Still don''t get it', 1)  
SELECT * FROM Movies
```

Update

After inserting you might want to change a record. Maybe you made a typo or new information is available. In those cases, you might want to update data.

Updating looks a lot like the INSERT, but you name the fields you want to change including the new data. Then you specify which record (or records) should get the new data.

Let's change the plot for Jaws, pretty simple. We need something unique, something only that movie has: The ID. The ID of Jaws is 3, in my table. Make sure you get the right one. The update query looks like this:

```
UPDATE Movies SET plot='When a killer shark unleashes chaos on a  
beach community off Cape Cod, it is up to a local sheriff, a marine  
biologist, and an old seafarer to hunt the beast down.' WHERE id=3
```

If you don't add the WHERE clause, all movies will be updated with

the same plot. But sometimes you want to update multiple records at once. Then you don't use the ID, but some other field that can act as a common factor. Let's say we want to mark all movies as seen then the rating is higher or equal to 7 and lower and equal to 9:

```
UPDATE Movies SET seen=1 WHERE rating <= 9 AND rating >=7
```

Delete

If you want to remove a record from the table you need to delete it (yes, remove and delete are not the same). Deleting a record isn't hard, but if you are not careful you might make a really big mistake.

The simplest form of a delete looks like this:

```
DELETE FROM Movies
```

This will delete **all** records in the table Movies and you don't want that. Let's say we want to remove the movie Inception. To avoid removing all the movies we need something unique that makes the movie Inception unique in the table Movies. And this is where the ID comes in... Again.

Just like a SELECT, we can add a WHERE clause to the DELETE, allowing us to pinpoint a certain record in the table. But we need to find the ID first. Use a SELECT to find the ID of the movie Inception and remember that one. Then execute the following query:

```
DELETE FROM Movies WHERE id=1003
```

Yes, funny thing: My IDs on my local SQL server jumped to around 1000. I have no idea why and searching for the problem isn't really helping. Some say it's because the SQL server wasn't shut down correctly, others say it's caching. This never happens on live SQL servers (Cloud for example), so I am not too worried about it.

Anyway, the query will delete a single record that has a unique ID of 1003. You can also delete more than one record with a WHERE clause. If you are cleaning up and want to delete all the movies you have already seen:

```
DELETE FROM Movies WHERE seen = 1
```

You can add filtering and searching as you please, but be careful and maybe you should use transactions if you are working with important data. Transactions are discussed later in this article.

Assignment

For this next bit to work, we need to change the database a bit. No worries, it will not be hard. But... I will let you do it on your own, mainly because I am done typing for now, so I will let you do some work.

All actions you need to take are described in the previous chapters.

First, create a new table with the name **Actors** and give it the following fields:

- Id (int, auto increment, primary key)
- Name (a text with 100 characters, does not allow null)
- YearOfBirth (a number, does not allow null)

Add the following records to the table:

- Keanu Reeves, 1964
- Mike Myers, 1963
- Jim Henson, 1936
- Jennifer Lawrence, 1990
- Roy Scheider, 1932

Next, add a new column to **Movies**. Name it **MainActorId**. It should be a number and it should allow NULL. Find the IDs of the actors that are the main actors for each movie and update the **MainActorId** accordingly. But you can't find any movie for Jennifer Lawrence, which is correct.

- The Matrix – MainActorId is the same id as the actor Keanu Reeves
- Shrek – MainActorId is the same id as the actor Mike Myers
- Jaws – MainActorId is the same id as the actor Roy Scheider
- The Muppets Take Manhattan – MainActorId is the same id as the actor Jim Henson (and no, Kermit is not a real actor!)

If you managed to make this work, continue to the next chapter.

Joins

There are situations where you want to combine two or more tables. If you follow the assignment above, you will have two tables: **Movies** and **Actors**. To combine tables you need references. The **Movies** table

has a reference to the Actors table; MainActorId. We could use that.

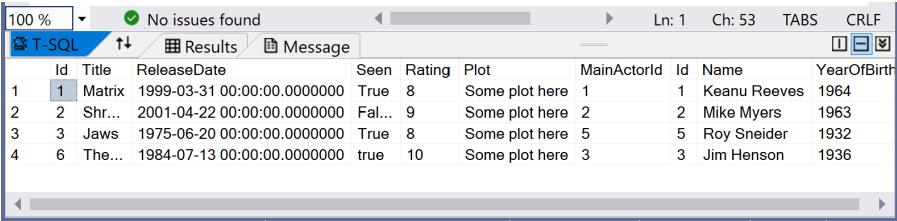
There are a few types of joins. I will start with the most common one, the inner join, to show what joins are.

Inner join

The inner join only shows the results when both tables match the join. Rows that don't have a match are not shown. Now, in our example not all joins are met; each movie has a MainActorId and all MainActorIds can be referenced to an actor. Except Inception. Let's create the query for the join:

```
SELECT * FROM Movies
INNER JOIN Actors ON Movies.mainActorId = Actors.id
```

I first do a selection on the main table, which is Movies. Then I inner join movies with the table Actors and tell them how to join them, by telling that Movies.MainActorId is the same as the Actors.Id. This results in the following grid:



100 % No issues found Ln: 1 Ch: 53 TABS CRLF											
T-SQL Results Message											
	Id	Title	ReleaseDate	Seen	Rating	Plot	MainActorId	Id	Name	YearOfBirth	
1	1	Matrix	1999-03-31 00:00:00.0000000	True	8	Some plot here	1	1	Keanu Reeves	1964	
2	2	Shr...	2001-04-22 00:00:00.0000000	Fal...	9	Some plot here	2	2	Mike Myers	1963	
3	3	Jaws	1975-06-20 00:00:00.0000000	True	8	Some plot here	5	5	Roy Sneider	1932	
4	6	The...	1984-07-13 00:00:00.0000000	true	10	Some plot here	3	3	Jim Henson	1936	

The table Movies is now expanded with the information on Actors, as long as the actor can be found. Thus, you don't see Inception.

Left join

This one joins the tables too but returns all the records from the left table and matching rows of the right table. But if there are no matching rows from the right table, the left table row is still showing, but the fields of the right table are NULL.

Inception still doesn't have an actor, but it will be shown since that is the way a left join works. The fields that normally hold the data for the actor will be empty.

And here is an example:

```
SELECT * FROM Movies
LEFT JOIN Actors ON Movies.MainActorId = Actors.Id
```

And the result:

100 % No issues found Ln: 1 Ch: 26 TABS CRLF											
T-SQL Results Message											
	Id	Title	ReleaseDate	Seen	Rating	Plot	MainActorId	Id	Name	YearOfBirth	
1	1	Matrix	1999-03-31 00:00:00.0000000	True	8	Some plot here	1	1	Keanu Reeves	1964	
2	2	Shrek The First	2001-04-22 00:00:00.0000000	False	9	Some plot here	2	2	Mike Myers	1963	
3	3	Jaws	1975-06-20 00:00:00.0000000	True	8	Some plot here	5	5	Roy Sneider	1932	
4	6	The Muppets Take Manhattan	1984-07-13 00:00:00.0000000	true	10	Some plot here	3	3	Jim Henson	1936	
5	7	Inception	2010-07-22 00:00:00.0000000	true	5	Some plot here	NULL	NULL	NULL	NULL	

Right join

I guess you already know: The opposite of the left join. All the records from the right table are shown and the matching records from the left table are shown. But if there are no matching records from the left table, NULL values are shown.

Example:

```
SELECT * FROM Movies
RIGHT JOIN Actors ON Movies.MainActorId = Actors.Id
```

And the result:

100 % No issues found Ln: 1 Ch: 27 TABS CRLF											
T-SQL Results Message											
	Id	Title	ReleaseDate	Seen	Rating	Plot	MainActorId	Id	Name	YearOfBirth	
1	1	Matrix	1999-03-31 00:00:00.0000000	True	8	Some plot here	1	1	Keanu Reeves	1964	
2	2	Shrek The First	2001-04-22 00:00:00.0000000	False	9	Some plot here	2	2	Mike Myers	1963	
3	6	The Muppets Take Manhattan	1984-07-13 00:00:00.0000000	true	10	Some plot here	3	3	Jim Henson	1936	
4	NULL	NULL	NULL	NULL	NULL	NULL	NULL	4	Jennifer Lawrence	1990	
5	3	Jaws	1975-06-20 00:00:00.0000000	True	8	Some plot here	5	5	Roy Sneider	1932	

Since Jennifer Lawrence isn't connected to a movie, she is listed in the results, but the values for a movie are all NULL. Inception isn't shown in the result.

Full join

In the previous examples, there was always missing an actor or a movie. The full join lists everything, not only when right or left has some form of connection. In a full join, if the right table doesn't have a match, the values are NULL. When the left table doesn't have a match, the values are NULL.

Example:

```
SELECT * FROM Movies
FULL JOIN Actors ON Movies.MainActorId = Actors.Id
```

And the result:

	Id	Title	ReleaseDate	Seen	Rating	Plot	MainActorId	Id	Name	YearOfBir
1	1	Matrix	1999-03-31 00:00:00.0000000	True	8	Some plot here	1	1	Keanu Reeves	1964
2	2	Shrek The First	2001-04-22 00:00:00.0000000	False	9	Some plot here	2	2	Mike Myers	1963
3	3	Jaws	1975-06-20 00:00:00.0000000	True	8	Some plot here	5	5	Roy Snieder	1932
4	6	The Muppets Take Manhattan	1984-07-13 00:00:00.0000000	true	10	Some plot here	3	3	Jim Henson	1936
5	7	Inception	2010-07-22 00:00:00.0000000	true	5	Some plot here	NULL	NULL	NULL	NULL
6	NULL	NULL	NULL	NULL	NULL	NULL	NULL	4	Jennifer Lawrence	1990

Aliases

As you might have noticed in the chapter about joins you might have duplicate columns. The Id for example is in Movies and Actors. If you want to get the Id you might get an error.

Msg 209, Level 16, State 1, Line 1
Ambiguous column name 'Id'.

Ambiguous column name 'Id'. That sounds correct since there are two columns with the name Id. There are two ways to fix this.

Column aliases

The easiest way is to explicitly name the column with the table name:

```
SELECT Movies.*, Actors.id
FROM Movies
FULL JOIN Actors ON Movies.mainActorId = Actors.id
```

The Movie.* means that I want to see all columns of the table Movies.

But, if you use this Entity Framework Core, for example, you still get the same error. We can temporarily rename a column in the query. We can give it an alias.

```
SELECT Movies.*, Actors.id AS actorId, Actors.name,
Actors.yearOfBirth
FROM Movies
FULL JOIN Actors ON Movies.mainActorId = Actors.id
```

When you execute this query you will see there are no ambiguous columns in the result.

Table aliases

Another reason to use aliases is to shorten the names of tables in the

query. Now, the names we are using aren't really long, but I will use those to give you an idea of what an alias table is:

```
SELECT * FROM Movies AS m
FULL JOIN Actors AS a ON m.mainActorId = a.id
WHERE a.yearOfBirth > 1960 AND m.Title LIKE '%e%'
ORDER BY m.title
```

One of the biggest benefits of this is that if the name Movies or Actors changes, you only have to change it in one place and don't have to walk through the whole query to find and replace Movies.

Transactions

Earlier we talked about the update and delete. We also established that it's pretty important to pinpoint those records you want to update or delete. It's pretty easy to delete or update all records if you are not careful. That is why it would be convenient to first try a query or multiple queries. This can be done with transactions.

Rollback

You can use transactions to execute SQL queries and roll them back if needed. A transaction is usually used to try out the queries, when they fail all operations inside the transaction are rolled back. If none fail, the transaction commits the results and outcome.

Here is a simple example of a transaction:

```
BEGIN TRANSACTION
DELETE FROM Movies
SELECT * FROM Movies -- Will show no movies at all
ROLLBACK TRANSACTION

SELECT * FROM Movies -- Will show all movies
```

If you execute this you will first begin a transaction, delete the movies, show no movies, and then show all the movies after the rollback. As you can see this is a great way to test out queries before you perform life-changing actions on the data.

Commit

If you are happy with your queries and you want to really execute them and not roll them back, you need to commit them instead of

using the rollback.

```
BEGIN TRANSACTION
DELETE FROM Movies
SELECT * FROM Movies -- Will show no movies at all
COMMIT TRANSACTION

SELECT * FROM Movies -- Will show no movies at all
```

Transactions are crucial for ensuring data integrity and consistency in the database, especially when you run multiple operations that need to be performed together.

If-Else

However, changing the ROLLBACK to COMMIT by hand all the time could be a bit time-consuming. When you have a SQL script you want to execute it and not change it all the time, right? It would be awesome to have some sort of if statement you can use to check if there are errors or not (...).

Well, there is. It looks a bit different than the C# variant, but the idea is the same. Besides that, SQL has a specific system variable. Well, there is more than one, but I am talking about the **@@ERROR**. This variable indicates if something went wrong or not. We could use that to check if we can safely commit the transaction or roll it back.

```
BEGIN TRANSACTION
DELETE FROM Movies
SELECT * FROM Movies

IF @@ERROR <> 0
BEGIN
    ROLLBACK
END
ELSE
BEGIN
    COMMIT
END
```

You start an If-statement with **IF** and then you give the condition. In this case, I want to check if the **@@ERROR** is not equal (**< >**) to **0**. If it is not 0 it means there is an error. If there is an error I want to roll back the transaction. And yes, you don't really need to put 'TRANSACTION' behind a rollback or commit. If the **@@ERROR** is 0, just commit the transaction.

Transactions are powerful things for complex database operations and

ensure we don't do stuff that could ruin the data inside the database.

Stored procedures

Okay, last part (I think, I am just writing as I go).

There are situations you have a few queries that need to run as a whole operation. You could write it and save the SQL script, open it later, and run it again. But that is not really convenient. If you are not at home, forgot your laptop, or whatever, you don't have that file with you, no software to open and run the SQL file.

SQL Server has something that allows you to store queries and execute them as a whole. We call these stored procedures. You can see a stored procedure as a method that is stored on the SQL Server and you can call and execute it.

To create a stored procedure you start a new query script and start typing:

```
CREATE PROCEDURE DeleteMovies
AS
Begin
    -- QUERIES HERE
END
```

CREATE indicates you want to create something, and PROCEDURE indicates you want to create a stored procedure. The DeleteMovies is the name of the stored procedure and you will use this to call and execute the procedure. Keep it simple, but make sure the name makes sense, just like in C#.

The AS indicates you are done with the parameter list (yes, you can add parameters) and read to BEGIN the queries.

Let's add the previous queries, the one with the transactions and deleting movies, in the body of the stored procedure. But let's change it a little bit so not all movies are deleted:

```
CREATE PROCEDURE DeleteMovies
AS
Begin
    BEGIN TRANSACTION
    DELETE FROM Movies WHERE seen = 1
    SELECT * FROM Movies

    IF @@ERROR <> 0
    BEGIN
```



```
ROLLBACK
END
ELSE
BEGIN
    COMMIT
END
END
```

Now it will only delete movies that are seen, leaving the unseen movies in the database.

If you execute this query, nothing will happen. Well, you will see a message saying Command(s) completed successfully. So something did happen. The stored procedure is created (hence the CREATE), but not executed. To execute it you need to run the following query:

Warning! Executing the query below will delete the movies that are seen. It is fine, but you will keep one movie stored in the table.

```
exec DeleteMovies
```

Exec stands for execute and then you give the name of the stored procedure you want to execute, DeleteMovies.

If you typed this under the CREATE PROCEDURE... etc... you might get an error:

Msg 2714, Level 16, State 3, Procedure DeleteMovies, Line 1

There is already an object named 'DeleteMovies' in the database.

This seems logical since you already created the stored procedure. What you can do is highlight the **exec DeleteMovies** and press the run button. This will only execute the highlighted text.

The stored procedure is now executed. It will not show you any results, except a message that the query is executed successfully.

A stored procedure can also have a SELECT, exposing data from tables. In that case, executing the stored procedure will show a grid, the same as when you do a normal SELECT, with the data.

Final words

Although Entity Framework Core takes away the need to write queries, you will always see something of queries or errors. It is imperative you know how to read SQL queries and how they work.

Stored procedures are super cool, but they are used less these days (in my experience, maybe others have a different opinion about this).

During your career, you will need to work with databases and sometimes Entity Framework Core will not always cope with complex situations. You will need to write an SQL query to make it work, which is fine.

Anyway, this was your introduction to databases and SQL. Especially the parts about queries, joining, and creating databases and tables are the most important chapters.

Chapter 3: Type-Along project

The SQL statements are easy to do in Visual Studio. I have explained how to create a database, table, and records. But the next chapters are all about code inside applications.

I have created a solution with some projects and code to make it easier for you to follow. This is the starting point of me explaining Entity Framework Core to you. We start with a basic code and build Entity Framework Core, and other code, inside this solution.

This chapter will explain the idea of the solution, the projects, models, interfaces, and some other background before we continue to the real stuff.

I will not explain how everything works because I will assume you have C# and .NET knowledge.

GitHub

You can find the source code on GitHub. You can easily clone or download the code and do whatever you want. You can find the repository here:

<https://github.com/KensLearningCurve/EntityFrameworkEBook.git>

You will see two branches:

- Main
This will contain the source code without Entity Framework Core. You can use this to insert the code I will be showing (and explaining) in the upcoming chapters.
- Completed
The completed branch will contain the code with Entity Framework Core and will be a representation of the complete product after you have finished this e-book. I would highly recommend using this branch when you are finished, not when you begin.

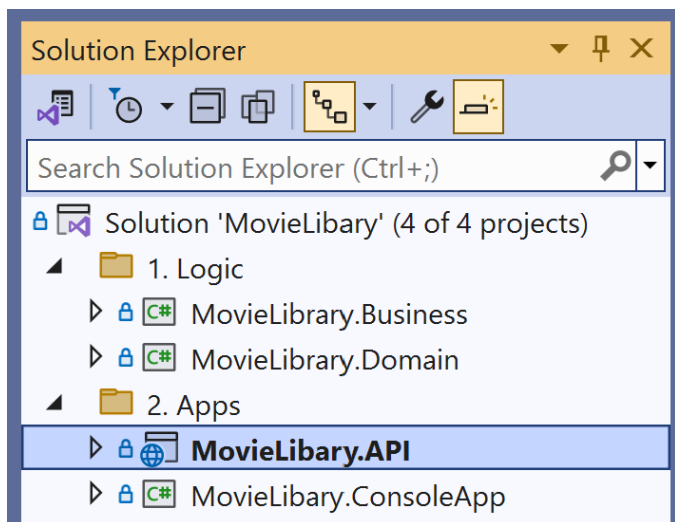
You can start by cloning or downloading the main branch and opening the solution. You might have to rebuild it to download and install the NuGet packages and build the files.

Let's take a quick walk through the different projects.

Projects

The solution has a few projects with each its own purpose. I have created folders to keep stuff separated.

All projects are .NET 8 and C# 12. I will be using Visual Studio 2022 (Preview) to build and run the projects. Note that you can also use most of the code examples in older .NET and C# versions. Maybe you need to use a slightly different syntax.



MovieLibrary.Business

This is the “brains” of the whole solution, which is also called the application layer. It contains all the logic that the presentation layers will need. We separate this to make sure that we can reuse the logic if we want to create a new application and don't need to copy-paste too much code.

The MovieLibrary.Business has one class, for now. This class is called Movies and only has logic for movies (single responsibility). You will see methods for retrieving, searching, adding, deleting, and updating movies.

The movies are now hardcoded in this class, which are placed on top. It's up to us to move these movies to a database using Entity Framework Core.

MovieLibrary.Domain

The *MovieLibrary.Domain* contains the models and interfaces (which are not many) that are being used throughout the solution in different projects.

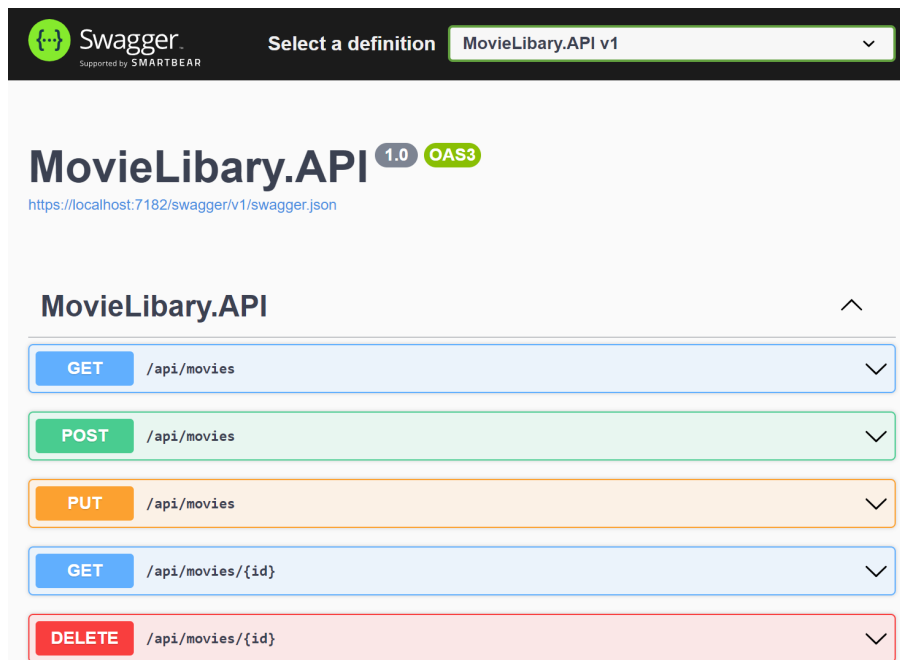
It contains a model for a movie, an actor, and an interface that is used in the Movies class in the *MovieLibrary.Business*.

MovieLibrary.API

This is a .NET 8 MVC minimal API and does what you can expect of an API; receive requests, handle them, and send back a response. It is configured with dependency injection, which has only the configuration for the (I)Movies.

We won't do much in this project, because all the logic is inside the *MovieLibrary.Business*, but we do need to configure Entity Framework Core here.

You can start the API and click around a bit. I have included Swagger, so you have some interface on the API.



Swagger
Supported by SMARTBEAR

Select a definition **MovieLibrary.API v1**

MovieLibrary.API 1.0 OAS3

<https://localhost:7182/swagger/v1/swagger.json>

MovieLibrary.API

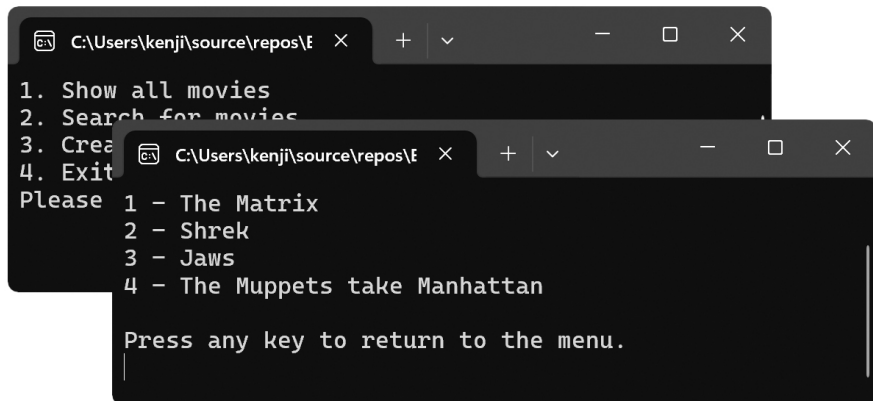
- GET** /api/movies
- POST** /api/movies
- PUT** /api/movies
- GET** /api/movies/{id}
- DELETE** /api/movies/{id}

MovieLibrary.ConsoleApp

A console application is more to the core; no fancy interface and pure C# and .NET. That's why I have included this one too. We will start our Entity Framework Core adventure with this application.

The console application will have a fraction of the Entity Framework implementation, but it should give you a good idea (and feeling) about the framework.

This console application has a simple menu that you can use. All logic goes through the *MovieLibrary.Business* so we don't have to change a lot of code here. Feel free to start it and press some buttons.



The image shows two overlapping console application windows. The top window displays a menu with four options: 1. Show all movies, 2. Search for movies, 3. Create new movie, and 4. Exit. The bottom window shows the result of selecting option 1, displaying a list of four movies: 1 - The Matrix, 2 - Shrek, 3 - Jaws, and 4 - The Muppets take Manhattan. Below the list, it prompts the user to 'Press any key to return to the menu.' Both windows have a title bar indicating the file path 'C:\Users\kenji\source\repos\E'.

```
C:\Users\kenji\source\repos\E x + v - □ x
1. Show all movies
2. Search for movies
3. Create new movie
4. Exit
Please 1 - The Matrix
      2 - Shrek
      3 - Jaws
      4 - The Muppets take Manhattan

Press any key to return to the menu.
|
```

Chapter 4: Entity Framework Core

Now that you know about databases using SQL, we can move further. If you have experience with ADO(.NET), you know there is much room for errors. Queries are just strings and a typo is easily made. And although you could fix it, the number of methods you need to do that same thing is just a lot.

With ADO(.NET) or pure SQL queries, the management of the database isn't really easy to do. Especially when you work in a team. Errors are easy to make and everyone could have a different version of the database.

Let's dive into something most C# developers use.

Welcome to Entity Framework Core!

In this chapter

We will (finally) look at **Entity Framework Core** and I will start with a simple introduction. You will learn about entities a little bit because **entities** will be handled in a different chapter further in this book.

We will also look at the differences between **code first** and **database first**. A small, theoretical introduction to both approaches. Please know that we will continue with the code first in the other chapters. Database first is not covered in this book!

Then it is time to talk about the **context**. Entity Framework Core starts with the **DbContext**; the bridge between your application and the database. It is here where you define your entities and tell Entity Framework Core what to expect.

Migrations are a big part of Entity Framework Core, so that's up next. How do they work? How do we create them? How do they keep our database structure up to date? Questions that will be answered here.

Then it is time to **implement Entity Framework Core** in the type-along solution. Everything inside this solution is hardcoded and we want to change that by using the database through Entity Framework

Core. This section is all about the code, not the explanation. I will zoom in on each aspect (and more) in the upcoming chapters.

So, don't feel threatened or lost after this chapter; It's just the beginning of our adventure.

Introducing Entity Framework Core

Developers, including yours truly, have been struggling with mapping data from a data source and code objects. But then ORM was introduced. ORM stands for **object-relational mapping** and enables us to work with domain objects and properties. The mapping struggle was over. I remember using Automapper and NHibernate a lot for my code. But then Microsoft released Entity Framework in 2008.

As usual, the newly released framework – which is built on ADO.NET – wasn't widely accepted. It had a lot of bugs and didn't do what was expected. In 2010, Microsoft released a new version, which was a bit better. They kept building on it and now it's one of the most common ORMs when you use Microsoft programming languages (C#, VB, etc). The current version is Entity Framework Core 8. Version 9 is scheduled for November 2024. However, in this article, I will use .NET 8 and Entity Framework 8.

Although the idea of Entity Framework and Entity Framework Core are the same, there are some big differences. Entity Framework Core is still tied to ADO.NET, but it has a more abstract and simple way of communicating with ADO.NET, making the performance better when using the core version.

Entity Framework is still tied to the Windows landscape, while Entity Framework Code can be used on different platforms.

But the biggest difference between both versions is the database providers. Entity Framework can communicate with Microsoft SQL Server (MSSQL). But Entity Framework Core can communicate with MSSQL, SQLite, PostgreSQL, MySQL, and many others. This is because Entity Framework Core is modular and easier to change by developers.

This e-book is written for Entity Framework **Core**.

As I said before, Entity Framework is built on ADO.NET but it has additional options. The mapping for instance is a very helpful feature.

Many aspects (especially errors) can be related to ADO.NET. It's a good practice to know a little bit about ADO so you know what is under the hood of Entity Framework Core.

Entities

Entities are a big thing when you use Entity Framework. Not because it's in the name of the framework, but how it works. An entity is usually an object in code that represents an entity in the data source. I like to say it represents a table in the database. The properties of the entity are used in the database as columns and settings. For example: If you have a movie entity, it has an Id (int), Title (string), and maybe a Rating (int). Entity Framework can translate these to an MSSQL table with a column for Id (int), a Title (varchar(500)), and a Rating (int).

Entity Framework Core can use these properties to build a table and map data from the database to the object and from the object to the database.

Database first vs code first

Entity Framework Core has two ways of managing your database. One is code first and the other one is the database first. There is a big difference, but code first is the most used. But before we dive in I want to explain both approaches.

Database first is used when there is already a database present and the database will not be managed by code. Code first is used when there is no current database, but you want to create one.

I like code first more because I can write entities (these are basically classes with properties) and let Entity Framework Core update the database accordingly.

It's just C# and I don't have to worry about the database much. I can create a class, tell Entity Framework Core it's an entity, update the database, and all is done!

Database first is the other way around. You let the database 'decide' what kind of entities you get. You create the database first and create your code accordingly.

This e-book will only cover the code-first approach. Database first

looks a lot like code first.

Packages

Entity Framework Core is a framework (it's in the name) and you don't have it by default inside your solution and projects. To ensure you have Entity Framework Core there where you need it, Microsoft has created a set of NuGet packages.

Each package has a different responsibility. Some packages will be used in one project and some will be used in several packages. It all depends on the requirements for your projects and application.

The package *Microsoft.EntityFrameworkCore* contains the basics of the framework. But before you can get data from a database, you need to tell Entity Framework Core what type of data source you are using. If you have an MSSQL database, you will need to install the package that is designed to handle such a database:

Microsoft.EntityFrameworkCore.SqlServer.

But if you have an Oracle database you don't need to install the package for MSSQL, but you will need the package *Oracle.EntityFrameworkCore*. This package isn't created by Microsoft, but it's built upon the Entity Framework Core and tweaked for usage on the Oracle database.

And there are more of these packages for different data sources. I will not explain them all, because that would mean this book would be way too big. But the basic idea of Entity Framework Core and the code you need will be the same, no matter what data source you use and the package you install.

Entity Framework Core supports a lot of data sources. Below you see a 'small' list of known data sources you can manage with Entity Framework Core.

- Microsoft SQL Server
- SQLite
- PostgreSQL
- MySQL
- MariaDB
- Oracle
- Cosmos DB
- Firebird
- InMemory

- Azure SQL Database
- Azure Cosmos DB
- Google BigQuery
- And much more.

Besides packages for the different data sources, you also need other packages. One package, the *Microsoft.EntityFrameworkCore.Tools*, allow you to manage the database structure and give you commands you can use in the CLI. More about this later.

Most packages will be discussed when we need them. You don't have to install the mentioned packages in this chapter. It's just to give you an idea of how Entity Framework Core is set up and what you can expect.

Context

With Entity Framework Core, it all starts with a context. It associates entities and relationships with an actual database. Entity Framework Core comes with **DbContext**, which is the context that we will be using in our code. In this section, we are going to discover the DbContext and use it in our solution.

DbContext

You can see the DbContext as a bridge between your application and the data source. It can connect to the data source and it can query data. It can also track changes you make to the entities in your applications and save those changes back to the data source.

The DbContext also knows how your data looks like and manages the structure of the data source.

It's a pretty nice tool that is the very beginning of using Entity Framework Core because we (usually) start with creating a class in our project that inherits from DbContext. It's in this class that we can store the connection string and tell Entity Framework Core which type of data source to use. But it is also used for seeding and setting up some connections and settings on and between tables.

Besides keeping track of the data, the DbContext can also map data from the data source to our C# models. Making it way easier to grab the information and work with it in our application(s).

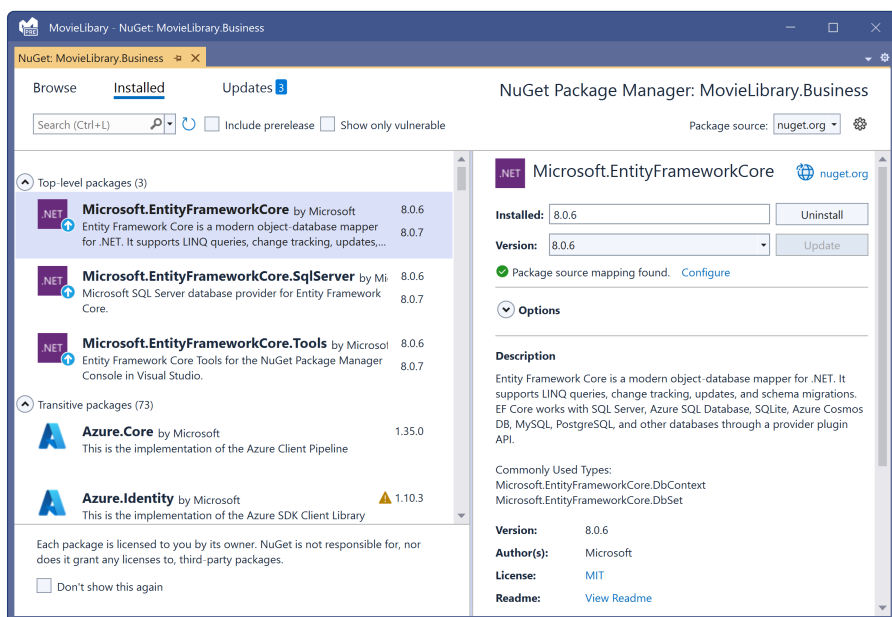
In short: The DbContext is the bridge between your application and the data source. It provides an API with a set provides a set of APIs to interact with the database, making it easier to perform common CRUD operations on the data.

The class

The first thing we need to do is create a context class. This is just a normal C# class with an inheritance from DbContext. That DbContext is a class of Entity Framework Core. We will create this class in the *MovieLibrary.Business* project. Create this class and call it *DataContext.cs*. Make sure it is public.

Next, we need the DbContext class from Entity Framework Core, but this is not a standard library. You will need to install a package before you can use it: [Microsoft.EntityFrameworkCore](#). This package contains all the classes and objects we need... For now.

To install this package you right-click on the MovieLibrary.Business project and select *Manage NuGet Packages*. This will show a new screen with a search field. Search for *Microsoft.EntityFrameworkCore*. Select the correct package from the list and click on the install button on the right.



When the installation is done a green checkbox will appear left of the selected package. You can now add the DbContext class to the

DataContext class.

```
using Microsoft.EntityFrameworkCore;

namespace MovieLibrary.Business;

public class DataContext : DbContext
{
}
```

Connection string

All we have to do now in this class is tell the DbContext which MSSQL server to use. This can be a local, cloud, or another server. Entity Framework Core needs a connection string to know which server to use.

Finding the connection string can be a bit tricky. But if you open the SQL Server Object Explorer, expand the SQL Server, expand the server (localdb\MYSSQLLocalDB), expand Databases, and click on the database (MoviesLibrary). Then press F4 or open the Properties window. This will show the database, server, and user. But expand the database now. If you take another look at the properties, you see a lot more information.

The most important property now is the Connection string. The value is a bit long. Select it all and copy it. Mine looks a bit like this:

```
Data Source = (localdb)\MSSQLLocalDB;Initial
Catalog = MoviesLibrary;Integrated Security = True;Connect
Timeout = 30;Encrypt = False;Trust Server Certificate = False;Application
Intent = ReadWrite;Multi Subnet Failover = False
```

The **Data Source** is the location of the database server. This could be a logical name, like in the connection string above, or an IP address. The **Initial Catalog** is the database you want to use. Queries don't include the database name (it could, but we don't), so Entity Framework Core needs to know which database you want to use.

Integrated Security is set to true, which means it will use Windows Authentication to access the database. There is also a **Connect Timeout**, which means that when it takes longer than 30 seconds to connect to the database, an error occurs.

We are not using a certificate to connect to the server (it's local), so

the **Trust Server Certificate** is set to false. We want to read and write to the database, thus the **Application Intent** is set to ReadWrite. If we only want to read, we can set it to Read-only.

I am not mirroring the database, so a failover is not needed. We can keep the **Multi Subnet Failover** on false. If you are working with subnets and you have multiple databases, this could be set to true.

Adding the connection string

To give the connection string to Entity Framework Core we need to override the **OnConfiguring** method from the DbContext. This OnConfiguring method is virtual, so we can override it. We use this method to configure the database and other options.

Overriding the OnConfiguring isn't that hard, but getting Entity Framework Core to give the connection string can be a bit tricky.

Entity Framework Core is not only used for MSSQL but also for other database structures, like Oracle and MySQL. We need to tell Entity Framework Core which database type we want to use by installing the right package.

In this case, we want to install a specific package from Nuget: [Microsoft.EntityFrameworkCore.SqlServer](#). It has all the functionalities to connect to an MSSQL database, read and write, and much more.

Overriding the OnConfiguring in the DataContext class and adding the following code:

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
    string connectionString = "Data Source=(localdb)\
\MSSQLLocalDB;" +
        "Initial Catalog=MoviesLibrary;" +
        "Integrated Security=True;" +
        "Connect Timeout=30;" +
        "Encrypt=False;" +
        "Trust Server Certificate=False;" +
        "Application Intent=ReadWrite;" +
        "Multi Subnet Failover=False";

    optionsBuilder.UseSqlServer(connectionString);
}
```

The **DbContextOptionsBuilder** has a direct link with Entity Framework Core and the settings of Entity Framework Core. By

installing the package *Microsoft.EntityFrameworkCore.SqlServer* we can tell this options builder we want to connect with an (MS)SQL server with the given connection string.

The base is now set and ready. Next up is the association with the entities.

Safety first!



There are better ways to store the connection string, but more about this in a later chapter.

DbSet

To tell Entity Framework Core what our entities are and how to store them in the database, we use **DbSet**. It's an Entity Framework Core property that will get the type of the entity and the name of the – in this case – table name. We want to store movie information in our database and we have a model for that: **Movie**. We can (re)use that as our entity. It looks like this:

```
public class DataContext : DbContext
{
    public DbSet<Movie> Movies { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        string connectionString = "Data Source=(localdb)\
\MSSQLLocalDB;" +
            "Initial Catalog=MoviesLibrary;" +
            "Integrated Security=True;" +
            "Connect Timeout=30;" +
            "Encrypt=False;" +
            "Trust Server Certificate=False;" +
            "Application Intent=ReadWrite;" +
            "Multi Subnet Failover=False";

        optionsBuilder.UseSqlServer(connectionString);
    }
}
```

I've declared a new property **Movies**. This will also become the table name in the database. The type of Movies is **DbSet<Movie>**, meaning I want to attach the object Movie as an entity to the context.

How this information will be used in the database is a topic for later in this book.

Migrations

Having a context class with a configured connection string to a database and an entity is nice and all, but we still need to create the actual database including the table (Movies). In the past, we had to create the database by hand, as we did in the previous chapter. But Entity Framework Core is capable of doing it for us. Since we are using the code-first construction we can create a script that will update our database with changes we made to the entities. We call these scripts migrations.

Migrations are nothing more than a translation from your code (entities) to the database. In our case, a migration will be a translation of the Movie model to the creation of an SQL table.

They are just C# code with commands to Entity Framework Core, telling what to do to the database. This is only to the database, not from the database to the code.

Another benefit of migrations is that these are cs-files and are stored on your local machine and, if you are working with it like I am, in git. You can share these migrations with others. This way everyone has the same database without the need to ship SQL scripts and hope everyone has executed them.

You create a migration with a command. You can execute this command with the Package Management Console, Developer PowerShell, or with a command prompt. I am going to use the **Package Manager Console** since this is integrated with Visual Studio.

Creating a migration

Creating a migration is simple. All you need to do is execute a command in the Package Manager Console and see the magic happens.

As you might have guessed... It's never that easy. Especially when you create a migration for the first time in your solution. Build let's just start from the beginning; with an error.

Creating a migration starts with the command **add-migration**,

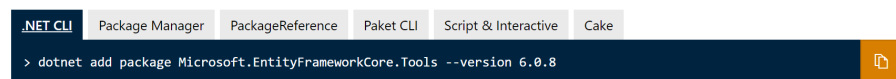
followed by the name of the migration. It looks like this:

```
add-migration Initial
```

If you execute this in the Package Manager Console, you will get an error:

add-migration : The term 'add-migration' is not recognized as the name of a cmdlet, function, script file, or operable program. Check the spelling of the name, or if a path was included, verify that the path is correct and try again.

No need to panic! We just need to install another package. Did I mention Entity Framework Core is mostly about installing packages? This time we need to install [Microsoft.EntityFrameworkCore.Tools](#). If you look at the description you'll notice it will install a few command line commands.



```
> dotnet add package Microsoft.EntityFrameworkCore.Tools --version 6.0.8
```

[README](#) Frameworks Dependencies Used By Versions

Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.

Enables these commonly used commands:

- Add-Migration
- Bundle-Migration
- Drop-Database
- Get-DbContext
- Get-Migration
- Optimize-DbContext
- Remove-Migration
- Scaffold-DbContext
- Script-Migration
- Update-Database

We won't be using all of them, but you see the **Add-Migration** here too and we need that one.

Make sure to install this package in the `MovieLibrary.Business!`

Before you execute the command again, there is another thing to point out: The default project. This is a setting in the Package Manager Console. You need to set this to the project that has the `DbContext` class. This is the `MovieLibrary.Business`.

When you are done, run the command again.

A few things happened in this order:

- Visual Studio will build the solution, checking if there are errors.
- Entity Framework Core will check if there is a snapshot of the database (which will be discussed later).
- Entity Framework Core will create a migration file, containing C# that will be used to update the database.

If you look carefully you will notice I didn't say the database will be updated. This add-migration is just a tool to create the changes, not execute them.

But if you look at the Solution Explorer, a few things have changed. A folder called *Migrations* has been added and it contains two files: `#DateTime#_Initial.cs` and `DataContextModelSnapshot.cs`.

The first one is the actual migration - changes that are not (yet) in the database - and the second one is a snapshot of our database as known in the code, thus not the real database.

But whatever you do; Never change these files unless you know what you are doing. Especially the snapshot. Change that and everything might go wrong.

Snapshots

But how does Entity Framework Core know what the changes are? By using a snapshot. After the add-migration is finished, a new folder has been created in your project. The folder *Migrations* will hold all the migrations you have created. These files can also be pushed to a Git repository.

If you open the snapshot file you will see a representation of the movies table and the properties of that table.

The more entities you use, the larger this file will become. In the end, the snapshot is the complete database based on your entities, not the database on the server.

Migrations

The other file, the migration itself, just has a representation of the changes you made to your entities or structure. This file contains only

the creation of Movies. It looks different than the snapshot.

```
namespace HelloWorld.Business.Migrations
{
    /// <inheritdoc />
    public partial class Initial : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder
migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Movies",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable:
false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Title = table.Column<string>(type:
"nvarchar(max)", nullable: false),
                    Plot = table.Column<string>(type:
"nvarchar(max)", nullable: false),
                    ReleaseDate = table.Column<DateTime>(type:
"datetime2", nullable: false),
                    Seen = table.Column<bool>(type: "bit",
nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Movies", x => x.Id);
                });
        }

        /// <inheritdoc />
        protected override void Down(MigrationBuilder
migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Movies");
        }
    }
}
```

There are two methods: **Up** and **Down**. The Up-method is when you want to update your database. The contents of this method will be translated into an SQL script and executed on the SQL server. You can see the **migrationBuilder** will create a table (CreateTable) with a name (Movies) and columns (Id, Title, Rating). Here you also see the types of those columns. Entity Framework Core has translated the string of C# into a **nvarchar(max)**.

If you have experience with SQL or read the previous chapter, you might see a lot of familiar things. Things like PrimaryKey, nullable, and the types. The auto-increment is here too, but it is called

SqlServer:Identity.

The Down-method will be executed if you want to reverse the migration or **downgrade** the database. In this case, it's simple: Just delete the table Movies, and the migration is made undone.

It's advisable not to change these methods. Simply because they are autogenerated and represent your entity **Movie**. Want to change something? Better create a new migration.

Updating the database

That's cool and all, but we still don't have a database. Correct! But the only thing you need to do is execute the following command:

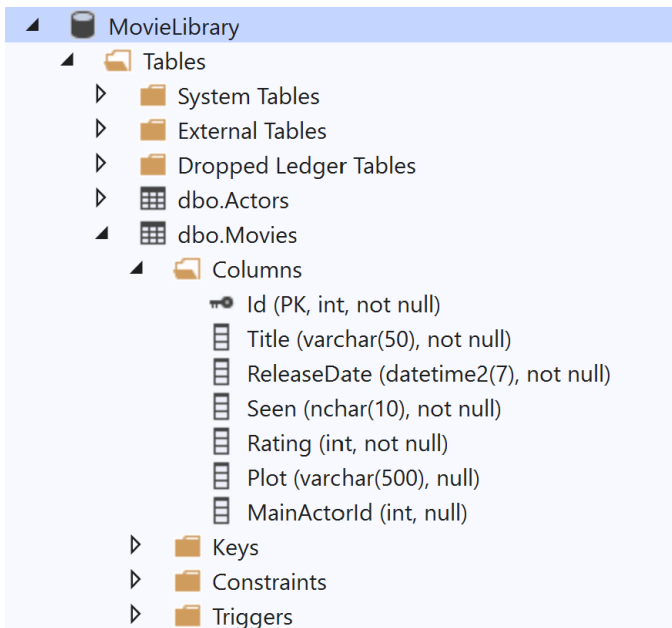
```
update-database
```

This command will check which migrations haven't been executed yet. By using the snapshot? Nope, by using the **EFMigrationsHistory** table in your database. But this table isn't available the first time. This is a hint for Entity Framework Core to create the database, create the EFMigrationsHistory, and execute all the migrations in order of creation date.

This EFMigrationsHistory keeps track of which migrations have been executed on the database. If you view the data in this table you will see just the initial migration. If you would create a new migration the initial migration will not be executed again, just the new migration.

If you delete all the records of the migrations history (don't!, Seriously, don't!) and update your database again it will execute all the migrations again, resulting in errors because the Movies table already exists.

Apart from history information, you also see the Movies-table. The properties (columns) are like those we used in the entity.



As said before; Another great feature of migrations is that you can exchange them with other developers in the team. Most of us use Git(Hub) to store code and other developers can grab that code too. Migrations are just files you can push to Git, let other developers fetch those files, and update their database with the migrations you created. This way you all have the same database structure. Something that could be an issue when you use plain SQL files and scripts.

Refactoring the code

We have our database and Entity Framework Core. All we have to do now is change the code of the service. We need to do a few things in the **MovieService** class:

- Initialize the context class.
- Refactor the methods in the **MovieService**.

We will start with the console application because I want to use dependency injection in the API, which is easier to show and explain.

Initializing the DataContext class

Maybe this is the easiest step. Let's create a private property in the **MovieService** that holds an initialized DataContext. Then we create a constructor that will set this property.

```
public class Movies : IMovies
{
    ...
    private readonly DataContext dataContext;

    public Movies()
    {
        dataContext = new();
    }
    ...
}
```



Can be done differently!

Yes, we could initialize the **dataContext** when we declare it, but it's a preparation for the next chapter.

The Get() method

Let's first change the **Get()** method. The property **dataContext** holds all the information, as well as information for the **Movies**, which is a public property. We can call the **Movies** property and that allows us to retrieve all the movies. But if we use **dataContext.Movies** it will return a **DbSet**, not a list of movies. To make it an official list, we need to cast the **dataContext.Movies** to a list:

```
public IEnumerable<Movie> Get() => dataContext.Movies.ToList();
```

But why? This has to do with the moment of execution. The **ToList()** executes a **SELECT * FROM Movies**. If we don't do that, the query will be generated at a later stage.

... But why? Well, imagine having 10.000 movies and you only want the movies you haven't seen (which are 10, for example). It is better to filter those movies on the SQL server, as I have explained in the previous chapter. So before you execute the actual query, you need to add a filter to it, like a **WHERE**. This is done with something we call **LINQ**. I would highly recommend you read up on **LINQ** since it will change your life! Don't worry about it too much, for now, we will cover this later.

We don't need a filter here, because the **Get()** method will return all movies.

But there are two other Get() methods. One is for getting a single movie by an ID and one to search for movies. Both work with a filter.

The Get(int id) method

This method needs to search for a single, unique movie. A movie with an ID and the ID is unique (primary key, auto-increment).

Thus, we need to use the LINQ **Single()** method, which has a lambda expression to filter the movies.

```
public Movie Get(int id) => dbContext.Movies.Single(x => x.Id == id);
```

The Single() does the same as the ToList(); it finalizes the query and executes it on the database server. It returns one Movie entity, or it throws an exception when no movie is found.

It will also throw an exception when the ID is 0 or less because there are no IDs with those values. It's a good idea to check for the value of the ID before sending it to the database. It will also throw an exception when more than one movie or no movies are found.

The Get(string query) method

Although this method will do the same thing as the Get(int id), it will search for the movies for a specific query. A query can be a part of the plot or title. A Single() won't work here, because more than 1 movie could be found.

We want to create something that looks like the SQL **WHERE** and the **LIKE**. Again, LINQ comes to help us out.

```
public IEnumerable<Movie> Get(string query) =>
    dbContext.Movies
        .Where(x => x.Title.Contains(query,
            StringComparison.OrdinalIgnoreCase)
            || x.Plot.Contains(query,
            StringComparison.OrdinalIgnoreCase))
        .ToList();
```

The changes to this method are minor; You only have to change **movies** to **dbContext.Movies** and add **.ToList()** at the end.

The Create() method

The Create(Movie movie) method is up next. Entity Framework has a great way of adding new data to a table:

```
public void Create(Movie movie)
{
    ArgumentNullException.ThrowIfNull(movie);

    dbContext.Movies.Add(movie);
    dbContext.SaveChanges();
}
```

But what is this **SaveChanges()**? Well, a bit the same as with the ToList(). Sending data, or deleting data, through Entity Framework Core is not executed as long as you don't save your changes. This way you can stack up the queries and execute them all at once. I actually dedicated a whole chapter to this, which is somewhat at the end of this book.

If you don't use the SaveChanges() after an update, delete, or add, the data is not submitted to the database.

The Delete() method

Deleting an entity is as simple as adding one. But you need to get the entity first. Entity Framework Core has something we call tracking. As soon as you get an entity from the table and store it in a C# entity, you have a connection. I will come back to this later.

So first, we get the single entity from the table and then delete it. Getting a single entity works the same as what we did in the **Get()** method, but with a filter:

```
public void Delete(int id)
{
    if (id <= 0) throw new ArgumentException($"The {nameof(id)} can't be 0 or less.");

    Movie movieToDelete = dbContext.Movies.Single(x => x.Id == id);
}
```

Single means that I want one item from a list. If none or more than one exists, I get an exception. I only expect one, since the ID is unique. Now we can tell the **dbContext** to delete this entity:

```
public void Delete(int id)
{
    if (id <= 0) throw new ArgumentException($"The {nameof(id)} can't be 0 or less.");
```



```
        Movie movieToDelete = dataContext.Movies.Single(x => x.Id ==
id);
        dataContext.SaveChanges();
    }
```

The Update() method

Last one. Updating works pretty simply: You get the entity first, then update the values of that entity, and you save the changes. A good practice is that you check if the ID of the movie, in the parameter list, is higher than 0. If not, chances are very big that the entity is not found.

```
public void Update(Movie movie)
{
    ArgumentNullException.ThrowIfNull(movie);

    if (movie.Id <= 0)
        throw new ArgumentException($"The {nameof(movie.Id)} can't
be 0 or less.");

    Movie movieToUpdate = dataContext.Movies.Single(x => x.Id ==
movie.Id);

    movieToUpdate.Seen = movie.Seen;
    movieToUpdate.Title = movie.Title;
    movieToUpdate.Plot = movie.Plot;

    dataContext.SaveChanges();
}
```

Because of the tracking, we can just update the values of certain properties of the entity and save the changes to the database.

Final words

And this is the very basics of Entity Framework Core. Don't worry! There is much more to learn and we just looked at the most logical part: CRUD.

In this chapter, you have learned how to set up a basic **DataContext** class with the **DbContext**, **DbSet**, and **OnConfiguring**. We have used a connection string to make sure Entity Framework Core can find the database.

We have connected our models to Entity Framework Core and created the database accordingly. **Migrations** are a great help to keep your

database up to date with the changes you make in the entities. The following chapters will create more migrations.

Using the DataContext is fairly easy; you initialize it and use it where and how you want. With **LINQ** you can get a single entity or a list, just don't forget to finalize the query with ToList(), Single, or other finalizers.

In the next chapters, we will dive more into the details of what we have done in this chapter.

Chapter 5: Setting up entities

Entities are an important part of Entity Framework. They represent the types and names in the database(s). It's a good idea to make sure you set up your entities as well as you can.

Things you should consider are the settings of the types (length of fields, default values, etc). But there is more to it. You can configure a lot to make sure Entity Framework handles your entities as you want it to handle them.

In this chapter

Entities are the most important part of the Entity Framework Core. Not only because it's in the framework's name, but it will also decide how the data will be stored inside your database.

Although they are essential, this chapter is rather small. It's not that entities are small and dumb, but they will also be shaped in the upcoming chapters.

Nevertheless, there are some specific subjects for entities. One of those subjects is **data annotations**. We can use data annotations to shape the fields in the tables. We can specify what the **maximum length** of a field is, set **requirements**, let the **database generate** the data, and more!

Some properties of an entity don't belong in a database table. We will be using the **[NotMapped]** attribute to keep the property, but not create the property in the table.

In the last section, we will look at how we can jump in when the state of an entity changes.

Contents of this chapter

Data annotations

In C#, data annotations are a way to add metadata to our classes, properties, and methods and are typically used to add validation rules

or provide information. They are attributes that you apply to your data model classes and their members.

Data annotations are not Entity Framework Core specific, but rather C# specific, and Entity Framework Core can work with it.

A good example of a data annotation used in C#, but works closely with Entity Framework, is the **MaxLength**, located in the namespace *System.ComponentModel.DataAnnotations*. It tells the property that it can only contain a maximum length. We can use that to set the maximum length in the column definition of the database.

And there are many more you can use. We will only look at the most important ones.

MaxLength

Let's continue with the **MaxLength** and put it to work. Data annotations are attributes on a class and/or property.

The property **Title** of the **Movie** class doesn't have this data annotation and that results in a column definition of *nvarchar(max)*. That type can contain 2,147,483,647 characters. Doesn't mean you need to add that many characters, but it is possible.

2,147,483,647 characters translated to gigabytes is 4GB... You can store up to 4GB in a column of the type *nvarchar(max)*. Not really ideal for a title.

The title should not be longer than 100 characters, which would be around 190 MB. A bit better, but still be careful.

To add this data annotation we set the attribute **MaxLength** on top of the property **Title**:

```
[MaxLength(100)]  
public string Title { get; set; }
```

This is a change to the database, so we need to create a migration to update our snapshot and the database.

We begin by adding a new migration with the following command:

```
add-migration SetMaxLengthMovieTitle
```

The new migration is created and the snapshot is updated. You don't have to look at it, it's there. Next, we update the database with the

proper command:

```
update-database
```

If you refresh the database in the SQL Server Object Explorer you will see that the Title is not `nvarchar(100)` and not `nvarchar(max)`:

The `nvarchar(max)` of the Title has changed to the `nvarchar(100)`.

Key

Each table in SQL needs a unique identifier, also known as the ID. In older versions, we had to tell Entity Framework Core which field is that ID. But Entity Framework Core uses naming convention and the data types to figure out which one is the primary identifier.

But in some rare cases, you need to tell Entity Framework Core which property of the entity is the ID. You can do that with the `[Key]` attribute.

```
[Key]
public int Id { get; set; }
```

You can also use the **DatabaseGenerated** on other properties, such as data-time.

Since Entity Framework Core already figured out that we have an ID, we don't have to change this to the database. You don't have to add the above to the Movie object. I just want to let you know about the Key attribute.

Column

It can happen you have created a column in your database, through a migration, and you found out the name is not correct. You could change the name of the property, create a new migration, and update your database, but you can also tell Entity Framework Core to get the data of a property from a different named column.

In our table **Movies** we have a column called Plot. It would be better to name it *Description*.

Instead of changing the name of the property and adding a new migration, we only change the property and add the attribute Column.

```
[Column("Plot")]
```

```
public string Description { get; set; }
```

There is no need to update the database because we haven't changed the structure. Except when you make a typo in the name of the column. The plot is different from the plot (lowercase). If you say “plot”, you will need to update the database.

Required

Required is a very popular data annotation. It's really powerful in making sure the value of the property is set. If the value is not set you will get an error.

```
[MaxLength(100)]  
[Required]  
public string Title { get; set; }
```

There is no need to update the database because we haven't changed the structure.

Add the **Required** attribute to the title.

DatabaseGenerated

We already seen this one with the Key-attribute and I already said you can do more with it than just set the identity seed on the id.

If you have a property that needs the current date and time. Think about a Create-field. Decorate that property with the DatabaseGenerated and Entity Framework Core will set the value for you.

```
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]  
public DateTime Created { get; set; }
```

The enum DatabaseGeneratedOption determines how the field will be set. Computed means according to the field type.

NotMapped

In some cases, you have a property inside your entity that is important but shouldn't be stored in your database. For example: You have a method in your C# code that can transform a string into a URL-friendly string. This means that the string can be used as a URL. You want to use this method on a movie title, making it URL-friendly so you can use it in the browser's URL.

```
public string? UrlFriendly { get; set; }
```

You don't have to store this transformed title, but you do need a property inside the `Movie` class. But if you add a property, Entity Framework Core wants to create a field for it inside the `Movies` table.

To make sure you can add a property without having to create it in the database, you can use the attribute **NotMapped**. This attribute will signal Entity Framework Core to ignore the property when sending or retrieving data from the database.

```
[NotMapped]  
public string? UrlFriendly { get; set; }
```

It's as simple as that! If you create a new migration, you will notice this field is not added to the database.

Assignment

Your turn to do something on your own. Follow the next steps to finish the assignment.

Connecting the actor

In the *MovieLibrary.Domain* is another model: **Actor**. This one should be in the database `Movies` as well. Follow the next steps:

- Make sure the property `Name` of the `Actor` can only hold 50 characters.
- Add the `Actor`-model to the `DataContext` class
- Add a migration so a table for the actors is inside the database
- Update the database

An actors-class

Next, create a new class in the `MovieLibrary.Business` project that has a single responsibility towards actors. It should have the same methods as the `Movies` class but specified for the `Actor`.

Don't forget to create an interface for this new class.

Act on states

In the previous section, we looked at the `DatabaseGenerated` attribute. Especially with the parameter `DatabaseGeneratedOption.Computed` a very handy data annotation that can set the current date and time when a new entity is created.

But what if you want to add another property named “`DateTimeUpdated`” which needs to be set each time the movie is updated? The `DatabaseGenerated` won't work, because that one is only for when a movie is created.

We need something that will be executed every time the state of an entity has changed and act on it.

Before we figure out how to do this, we add the property **`DateTimeUpdated`** to the `Movie`.

The `ChangeTracker`

With Entity Framework Core comes the **`ChangeTracker`**. This does what you expect; track the changes. This also comes with an event handler called **`StateChanged`**. You can use this to execute your code as soon as the state of an entity changes.

States are modified, added, and deleted.

To use the **`ChangeTracker.StateChanged`** you need to add it to the constructor of the class that inherits the `DbContext`; the `DataContext`:

```
public DataContext(DbContextOptions<DataContext> options) :
    base(options)
{
    ChangeTracker.StateChanged += ChangeTracker_StateChanged;
}
```

This will also create the following private method if you use the tab-function:

```
private void ChangeTracker_StateChanged(object? sender,
    EntityStateChangedEventArgs e)
{
    throw new NotImplementedException();
}
```

All we have to do is add the code that will check if the entity is `Movie` and which state the entity has and act on it:

```
private void ChangeTracker_StateChanged(object? sender,
    EntityStateChangedEventArgs e)
{

```



```

        if (e.Entry.Entity is Movie movieEntity)
        {
            if (e.Entry.State == EntityState.Modified)
            {
                movieEntity.DateTimeUpdated = DateTime.Now;
            }
        }
    }
}

```

This will update the value for the property `DateTimeUpdated` each time you update an entity of the type `Movie`.

Better approach: Interfaces

Chances are that you have more entities that have a property like `DateTimeUpdated`. To make sure you don't have to check if the current entity in the `ChangeTracker` is one of those types, you can use an interface.

I see an interface as a blueprint of a class: The properties and methods defined in an interface have to be in the derived class. This means that if we add `DateTimeUpdated` in an interface and make the entities inherit from this interface, all entities will get the property `DateTimeUpdated`.

Let's create a simple interface inside the folder *Interfaces* of the *MovieLibrary.Domain*. Add the property `DateTimeUpdated` property:

```

public interface IBaseEntity
{
    DateTime DateTimeUpdated { get; set; }
}

```

Next, connect this interface with inheritance to the entities. Here is the `Actor` as an example:

```

public class Actor : IBaseEntity
{
    public int Id { get; set; }
    [MaxLength(50)]
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    public DateTime DateTimeUpdated { get; set; }
}

```

In method `ChangeTracker_StateChanged`, we don't have to check on `Movie`, but on `IBaseEntity`. This means that all entities that have this interface also have the property `DateTimeUpdated` and can be updated with the current date and time.

```
private void ChangeTracker_StateChanged(object? sender,
EntityStateChangedEventArgs e)
{
    if (e.Entry.Entity is IBaseEntity baseEntity)
    {
        if (e.Entry.State == EntityState.Modified)
        {
            baseEntity.DateTimeUpdated = DateTime.Now;
        }
    }
}
```

This is a cleaner and better approach for tracking the state of an entity and acting on it.

Final words

Entities are a big part of Entity Framework Core and they should be kept clean and simple. They are basically classes with properties. Each entity represents a table in the database and each property of an entity represents a field of that table.

Because they are important and have a big influence on the database we need to configure them with **MaxLength**, making sure the field doesn't get too much information which makes the database grow in size.

We also look at the **Key**, **Required**, **NotMapped**, and **Column**. Data annotations that help set up the database table. You can use **DatabaseGenerated** to let Entity Framework Core generate the value for you.

When an entity **changes its state**, like when you want to add or edit the entity, you can attach your own even so you can do something when the state of an entity is specific. An example is to set the date and time when an entity is modified. Instead of doing this in the update method, you can automate this with the ChangeTracker.

Chapter 6: Manage Migrations

Migrations are a big part of Entity Framework Core. They keep the structure and some parts of the data up to date. They help you submit changes from your C# code to your database.

Another big advantage of these migrations is that they are easy to share with others, with team members for example. Because the migrations are files they can be committed to Git. Others can fetch those files and update their local database. This way everyone has the same database—no more human errors with update-instructions.

Managing migrations is something different. You need to know when you need to create, remove, or delete one. That's what this chapter is all about.

In this chapter

Migrations are a big part of Entity Framework Core and they help you keep your database structure up to date.

In this chapter, we will learn how to **create migrations** and how to send them to the database. You already did this, but now you get a bit more information about the process.

If you make a mistake in your migrations, you can revert them. **Reverting and/or deleting a migration** isn't as simple as just deleting the migration file. You will have to revert the migration before you can delete it. You need to make sure your database and the **snapshot** are consistent and always up to date. Deleting a migration file will disrupt this flow.

Although the migrations are auto-generated, you can still **add** some of your **own SQL** experience. Changing a migration is possible but you have to be careful not to change the reason why the migration was created in the first place.

Some data in your database is pretty static; it won't change. Or you have data that has to be present, like a list of countries. For this, we can use **seeding**. This is a process where you seed the data into your database through migrations.

Another reason to use seeding is to add testing data.

Need to create and send your migrations to a server that doesn't support migrations? No worries, we will learn how we can **script the migrations** to an SQL file. Entity Framework Core will translate the migrations to SQL, which you can execute on the server if needed.

Is your database a real mess with all kinds of test data or something else is wrong? Maybe it's time to **start over** and let Entity Framework Core delete the complete database and execute all the migrations from the start.

Contents of this chapter

Reverting Migrations

Usually, when you want to remove something you delete it. You can simply delete a file or remove the text/code. Migrations with Entity Framework Core are basically both. But deleting the migration files isn't enough. The migrations are still in your database. To remove a migration you also need to downgrade your database.

Remember both methods inside a migrations? One is `Up()` for pushing the changes from your code to the database and the other one is `Down()` for undoing those changes in your database.

We just made a migration to set the max length on the Title column. It changes the `nvarchar(max)` to `nvarchar(100)`. But 100 is maybe a bit too large. Maybe 50 is better. Now, instead of changing the 100 to 50 in the entity, creating a new migration, and updating the database, it might be better to undo the migration to `nvarchar(100)` and create a new migration for the 50.

Removing a migration works only from the newest to the oldest. You can't just remove the 3rd migration of 5 migrations in total. To remove a migration you need to follow 2 steps:

- Update the database to a past migration
- Remove the migration

Update database to the past

The first thing we need to do is make sure the changes from the

migration-to-be-removed are made undone. This needs to be done to update the database to a migration before the migration we want to remove.

In short: We need to update the database back to the **Initial** migration. This will trigger the **Down()** method of the **SetMaxLengthMovieTitle** migration.

```
update-database Initial
```

By specifying a migration, the name only, you can go back to that migration. Update-database without the migration name is updating the database forward.

After you have executed the above command, you will see that the **nvarchar(100)** is set back to **nvarchar(max)**.

If we have 15 migrations and downgrade the database to migration number 5, the 10 Down methods in the migrations after the 5th are all executed. This might be a good hint to really think about the entities and settings that influence your database.

Remove the migration file

But the migration file (SetMaxLengthMovieTitle) is still there. In fact, while the Title column of the table Movies says it's a **nvarchar(max)**, the DataContextModelSnapshot still says the Title is a **nvarchar(100)**. This will be fixed when we remove the migration file.

We can't just delete it by selecting it and delete it from our disc. If we did that the DataContextModelSnapshot won't be updated. Instead, we need to execute another command:

```
remove-migration
```

This command will delete the migration that isn't executed on the database and update the DataContextModelSnapshot.

You can also use the remove-migration command when you create a migration and realize you made a mistake before you send it to the database. After creating the migration the DataContextModelSnapshot is updated and you need to undo that.

Bottom line: Never delete migrations by hand; use the command line.

Change the MaxLength of the Title to 50 and create a new migration

for it. **Do not update the database!**

Adding extra information

Migrations are very powerful but even they can use a little help sometimes. Earlier I told you not to change the contents of a migration file, but in some cases, you can and should.

In some rare cases, you need to update data, store information temporarily to use later after the migration, or execute a stored procedure.

Or you made a mistake in the earlier seeded data and you create an empty migration, add the SQL, and update the database with the migration. It's all possible.

Adding SQL

Let's assume the rating is now based on 1 to 5, but we want to change it to 1 to 10. That means that the current ratings have to be multiplied by 2. If you want to execute this with SQL the query would look like this:

```
UPDATE Movies SET Rating = Rating * 2
```

We can add this query to a migration. You can use the previous migration and add the **modelBuilder.Sql()** method. This method has a parameter **string sql**, which is the query as shown above. It is used to execute pure SQL queries and can be inside the Up() and Down() methods.

Place the modelBuilder.Sql() in both the Up and Down methods and the query will be executed when you update or downgrade the database.

When we add it to the migration it looks like this:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AlterColumn<string>(
        name: "Title",
        table: "Movies",
        type: "nvarchar(50)",
        maxLength: 50,
        nullable: false,
        oldClrType: typeof(string),
        oldType: "nvarchar(max)");
}
```

```
    migrationBuilder.Sql("UPDATE Movies SET Rating=Rating * 2");  
}
```

Let's add the opposite to the Down() method:

```
protected override void Up(MigrationBuilder migrationBuilder)  
{  
    migrationBuilder.AlterColumn<string>(  
        name: "Title",  
        table: "Movies",  
        type: "nvarchar(50)",  
        maxLength: 50,  
        nullable: false,  
        oldClrType: typeof(string),  
        oldType: "nvarchar(max)");  
  
    migrationBuilder.Sql("UPDATE Movies SET Rating=Rating / 2");  
}
```

You can now update the database. It will update the Title field, but also update the ratings by multiplying it with 2.

Adding migrations

We already created some migrations, so there are no big surprises there. In short: When you want to update your database schema (the structure), you make a migration. After that, you can update the database.

When you create a migration for the first time, Entity Framework Core will also make a snapshot of the database. This snapshot is not from the actual database, but from the database according to your local files; the DbContext with the DbSet, and other settings.

Then you update the database. Entity Framework Core will create the database if it doesn't exist yet. It will then execute the migrations that haven't been executed yet.

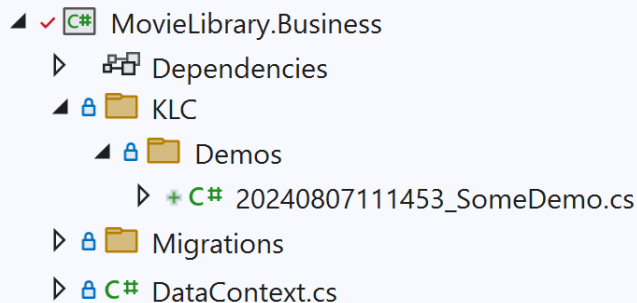
If you already executed migrations before, the database will contain a table called **EFMigrationsHistory** and this keeps track of which migrations have been executed on the database. If you view the data in this table you will see all the past executed migrations. When you create a new migration those older migrations will not be executed again, just the new migration. If you delete all the records of the migrations history (don't!) and update your database again it will execute all the migrations again, resulting in errors because the changes are already in your database.

OutputDir

All the migrations you create are stored in the folder *Migrations* in the project that holds the class inheriting DbContext. If you want to change that folder, use the following command when creating a new migration:

```
add-migration SomeDemo -OutputDir KLC\Demos
```

This will result in the following folder structure of the *MovieLibrary.Business*:



This is a great way to order your migrations as you please. You can split your migrations in configuration, changes on tables, etc.

Stored procedures

In an upcoming chapter, we will learn how to use stored procedures with Entity Framework Core. These stored procedures are somewhat methods in the database and they are not entities. You can't create a migration for them. Well, not automatically.

Because Entity Framework Core can't create stored procedures, we need to create those by hand. But it's not a good idea to store them in the database only. It defeats the idea of migrations with Entity Framework Core.

To create a migration for a stored procedure, we need to create an empty migration, add the SQL ourselves, and update the database.

Look at the following SQL:

```
CREATE PROCEDURE GetMovies
AS
BEGIN
    SELECT
```



```

        Title,
    CASE
        WHEN seen = 1 THEN 'Yes'
        WHEN seen = 0 THEN 'No'
    END As Seen,
    CASE
        WHEN Rating BETWEEN 1 AND 2 THEN 'Bad'
        WHEN Rating BETWEEN 3 AND 4 THEN 'Okay'
        WHEN Rating = 5 THEN 'Good'
        ELSE 'Unknown'
    END AS Classification
FROM
    Movies;

END

```

The stored procedure isn't exceptional and we don't want it to be special or complex. We'd like to know how we use Entity Framework Core to execute a stored procedure.

All this stored procedure does is get all the movies with some modifications on the fields. Instead of 1 or 0, I want to retrieve 'yes' or 'no' if the movie is seen or not. The rating is also set with words instead of numbers. If the rating is not 1 to 5, it's classified as an unknown rating.

We want to create this stored procedure in our database through a migration. To do this we create an empty migration by adding a migration, even though we have no changes in the DataContext or the entities.

Execute the following command in the Package Manager Console, or other command line if you prefer:

```
add-migration AddSPGetMovies
```

This gives you an empty migration with the methods **Up()** and **Down()**. We then use the **migrationBuilder.Sql** to execute an SQL query on the database. This SQL query is the whole stored procedure as shown earlier:

```

protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql(@"CREATE PROCEDURE GetMovies
        AS
        BEGIN
            SELECT
                Title,
                CASE
                    WHEN seen = 1 THEN 'Yes'
                    WHEN seen = 0 THEN 'No'
                END As Seen,
                CASE

```

```

        WHEN Rating BETWEEN 1 AND 2 THEN 'Bad'
        WHEN Rating BETWEEN 3 AND 4 THEN 'Okay'
        WHEN Rating = 5 THEN 'Good'
        ELSE 'Unknown'
    END AS Classification
FROM
    Movies;
END ");
}

```

This will create the stored procedure when you update the database. But we also need a down, just in case you might want to delete the migration. The Down() should delete the stored procedure:

```

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("DROP PROCEDURE GetMovies");
}

```

You can now update the database and the stored procedure is created in the database. You also have a migration for this stored procedure, which makes it easier to distribute to team members or environments.

Seeding

As you just learned you can add custom SQL to the migrations. But don't start using this to add data to your database tables. Adding data by migrations is possible, but we rather do this by something that is called **seeding**.

Seeding is a process where you define the data for a database table and use a migration to send that data to the database.

A reason to use seeding can be that some data is always the same. Data like cities, countries, genders, and other options. But you still want them in your database. You should also add some test data to your database as soon as you update or create your database with Entity Framework Core.

Or when a new coworker gets the code from Git he/she has to update the database and has a fully working environment because some data is already in the database.

Overriding OnModelCreating

Seeds are created in the DataContext (or the class connected to the DbContext). You can override a method from the DbContext called

OnModelCreating. Seeding is one of the functions of this method.

Override the method in the DataContext class now.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
}
```

The **ModelBuilder** is a class that you can use to configure entity data. With the **ModelBuilder** you select what you want to do on a specific entity. It is also this class that we need to use to create seeding.

The **ModelBuilder** is not only there to seed your database, but also to set up different relations and settings of your entity. We will discover more about the **ModelBuilder** in upcoming chapters.

HasData

Before can start seeding, we need to select the entity. From there we can configure that entity or add seeding. Selecting an entity is pretty simple:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Movie>()
}
```

Everything we do behind this line will affect the **Movie** entity.

Now I can use the **HasData** method. This method is specifically for seeding. It tells Entity Framework Core that this entity (**Movie**) has data and that data is specified in the parameters. I can now create a new **Movie** and set the properties:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Movie>().HasData(new Movie()
    {
        Id = 1,
        Description = "Katniss volunteers to replace her sister in
a tournament that " +
            "ends only when one participant remains. Pitted
against contestants " +
            "who have trained for this all their life, she has
little to rely on.",
        ReleaseDate = new DateTime(2012, 03, 21),
        Seen = false,
        Title = "The Hunger Games"
    });
}
```

Yes, we have to set the ID. When this seed is transformed into a migration, the identity seed is turned off, which is normal.

The downside is that you need to know which value you can use. When there is already data in the table you could run into problems. Therefore, it's crucial and advisable to create the seeds as soon as you make the table.

If you want to create more seeds for the movie, you just duplicate that piece of code and change the values of the Movie properties.

The migration

We can now create the migration. This works the same way as when we added the entity or changed the entity.

```
add-migration SeedingMovies
```

When we look inside the migration, we see something similar to the code below.

```
0 references
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.InsertData(
        table: "Movies",
        columns: new[] { "Id", "Plot", "Rating", "ReleaseDate", "Seen", "Title" },
        values: new object[] { 1, "Katniss volunteers to replace her sister in a tournament " +
            "that ends only when one participant remains. Pitted against contestants who have " +
            "trained for this all their life, she has little to rely on.",
            0, new DateTime(2012, 3, 21, 0, 0, 0, DateTimeKind.Unspecified), false, "The Hunger Games" });
}

0 references
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DeleteData(
        table: "Movies",
        keyColumn: "Id",
        keyValue: 1);
}
```

As with all migrations, we have an Up() and a Down() method. The Up() method will insert the data. And the Down method will delete the data.

You can now update the database. When you view the data of the Movies table you will see the movie "The Hunger Games".

If you get an error hinting that the ID is already used, use a different ID.

Extras

Now that you know the most needed functionalities of migrations, I have some extras for you. You won't need these things a lot (if ever), but it's good to know they are possible. And maybe you might run into the need to use them

Scripting migrations

If you have a neat migration setup and your database is up-to-date it's pretty annoying when you have an environment you can't use your migrations to update that database. These servers (still) exist and they are usually not reachable by 3rd party applications such as Visual Studio or Entity Framework Core.

To make sure you can still create the database, with all the tables and seeding, on that server, you can do two things: Create everything by hand or let Entity Framework Core create an SQL script. I would choose the latter and let the SQL script be generated by a tool. And Entity Framework Core is such a tool.

Note: You can generate an SQL script from your migrations, but only the migrations that are in your `DataContextModelSnapshot`, not your database.

Entity Framework Core has a command to create an SQL script that has all the information from the migrations and converts it to SQL.

To create such a script, simply execute the following command:

```
script-migration
```

This will create and open an SQL file that represents all the migrations, but in SQL.

These files are stored in your .NET version folder of the Debug folder. In our case:

EntityFrameworkEBook\MovieLibrary.Business\obj\Debug\net8.0

This folder is in the gitignore, so it will not be committed to git.

Starting over

When you develop your application - and your database - you might get some pollution in the database. A lot of test data is getting in the way. Or maybe you made a mistake in your tables or database and

you want to start over.

Don't worry! You can do that. There is a simple command to drop the database, allowing you to create a new database and execute all the migrations in your project.

Do know that this command will delete everything in your database and there is no CTRL + Z possibility! Use it with care!

The command is this one:

```
drop-database
```

Sounds logical, right? You do have to confirm you want to delete the database. Just in case you wanted to type update-database, but typed drop-database (...).

When it's done removing the database, you can update your database and you have a nice, clean database.

List migrations

The last part for the extras.

If you have some migrations you might want to list them to see which migrations you have, in order of creation, and which are applied in the database and which are not.

Such a list could also be used to document which migrations you have.

To create such a list you need a simple command:

```
get-migration
```

The result is a table, in the output, with all the current migrations and the states (applied yes/no).

The **id** is the part you see in the files of the projects, under the folder Migrations. The **name** and the **safe name** are the names you gave to the migrations but without the date and time. And the **applied** is true or false. True when the migration is applied to the database, false when not.

Assignment

Your turn to do something on your own. Follow the next steps to finish the assignment.

Adding seeding

Add all the movies that are listed in the Movies class (the `List<Movie>`) in the seeding.

When you are done adding them, you are going to reverse the migrations in a way that the previous migration is removed. I am talking about the migration **SeedingMovies**. Make sure you delete migration as shown in this chapter

Then you create a new migration with the same name: **SeedingMovies**. Make sure that all the movies, including *The Hunger Games*, are in the migration.

Before you continue, remove the database entirely in the correct way; using the command line.

After you have deleted the database, update the database with all the migrations. You should have a nice and clean database with only the movies in the seeding-migration.

Seeding actors

The Actors table has a minor issue: A year of birth is not easy to work with and calculating the age with just a year isn't easy. We have to change that field to a date. Since we have new migrations after the creation of Actors, we can't just remove them.

Change the property **YearOfBirth** to **DateOfBirth** and change the type to **DateTime**. Create a migration for this, but don't update the database yet.

next, add seeding to the Actor's table as we learned in this chapter. Add the following seed information:

- Keanu Reeves September 2, 1964
- Mike Myers May 25, 1963
- Jim Henson September 24, 1936
- Jennifer Lawrence August 15, 1990

Create a migration for the seeding. You now have 2 migrations. Update the database.

Final words

Migrations help you to keep your database up to date by translating your C# code to SQL. In this chapter, you have learned how to **revert a migration** by using a command to make sure you safely downgrade the database and remove the file.

Adding a new migration is already known, but in this chapter, you have learned how you can **add custom SQL** to an existing migration and update the database with that custom SQL. You also learned how to **change the directory** where the migration files are stored inside your solution and how you can add stored procedures to a migration.

We also discussed **seeding**. A technique we use to insert data through a migration in the database. Ideal for data that doesn't change much or for test data. We don't have to add data by hand every time we create a new database.

In the last section, I showed you some extras like the scripting of migrations, the drop-database, and how to list the current migrations.

Chapter 7: Relationships

In the chapter *It all starts with a (SQL) database* subchapter Joins, we looked at how to connect different data from different tables. We call these relationships or related data.

Entity Framework Core is also capable of joining these pieces of data together and it will be translated to the SQL joins.

Joins, or related data, are a very important part of the database structure. In this chapter, we are going to add genres and connect the movies to these genres. We will also look at the actors playing in the movies.

Each movie has one genre, but multiple actors.

In this chapter

Commonly, you have data located in different tables. In some cases, you want to combine different tables in one result. We do this with **relationships** and it's adding joins to the query. But Entity Framework Core doesn't have joins in C# code, it has different ways to add the joins.

Before we start joining the different entities, we will look at **conventions** and **navigations**. These are two important aspects of (auto) joining the entities.

Then we will look at two different types of relationships: one-to-one and one-to-many. We will use the Movie -> Genre for a **one-to-one relationship** since a movie has one genre. There are three ways Entity Framework Core can grab related data: **eager**, **lazy**, and **explicit**. We will look at all three of them.

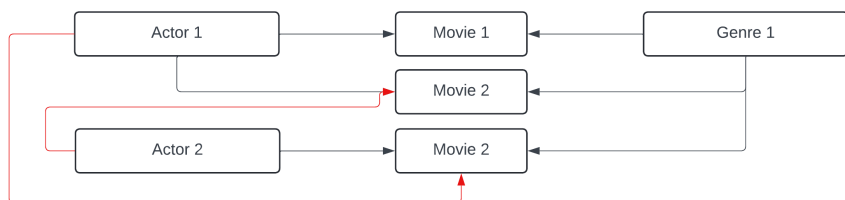
A movie has more than one actor, thus we will use the Movie -> Actor entity for a **one-to-many relationship**.

Theory

One of the things you want to do while using databases is join two or more tables into one query, getting a new table with all kinds of extra

information. Let's say we have our movies and we want to join a genre into it. Or we want to join the actors in a movie. These pieces of information are often in a different table. We call this type of data related data.

But why not put them all in the same table? Simple: An actor plays in different movies. To avoid recreating the same actor over and over again we can reuse his/her information for each new movie he/she plays in. The same goes for a genre.



As the image above shows, one movie has one or more actors. But a movie can only have one genre.

The relationship between the movies and the actors is a **one-to-many** relationship: one entity can have a relationship with multiple other entities.

The relationship between the movies and genres is a **one-to-one** relationship: An entity can only have one relationship with another entity.

Conventions

Entity Framework Core does a lot with conventions, which is good for us. This means less work because if we do it right, Entity Framework Core will figure it out for us.

The primary key and auto-increment is an example. In the past, we had to tell Entity Framework Core that the ID field is the primary key and has auto-increment. But now Entity Framework Core knows that a field with the name ID is just that and we don't have to mark it anymore.

The same goes for relationships between entities. If we do it right, Entity Framework will create a lot of logic to create those relationships.

Navigations

The relations between data are done with **foreign keys**. A foreign key is like a link between the entities. For example, the **GenreId** is the link between the Genre in the Movie and the Id of the Genre.

Entity Framework Core uses relationship navigation to connect entities and navigate from one entity to another, using defined relationships.

Relationship navigation in Entity Framework Core refers to how entities (objects) are connected and how we can navigate from one entity to another using their defined relationships. It allows us to access related entities through properties defined in our entity classes, making it easy to pass through and retrieve data across connected tables in a database.

There are three navigation types we use and know:

Reference navigations

A reference navigation is the “one” part of the one-to-many relationship and it's a simple object reference to another entity. For example: A movie has one genre. Inside the movie, we navigate to one genre. The genre is the reference navigation.

Entity Framework Core will use conventions to create this navigation on its own when the type is public, and has a getter and setter, is an entity, not static, and not an indexed property.

Collection navigations

This type is the “many” part where you reference a list of entities in the “one”. An example of the collection navigation is the actors of a movie. A movie has more than one actor. The reference from the movie to the actors is the collection navigation.

Convention can be applied to this one as long as the type is public, has a getter and setter, it's an IEnumerable (or derived), is not static, and is not an indexer property.

Configuring navigations

Both the reference and collection navigation use the Entity Framework

Core conventions. But if you can't, because of various reasons, you need to configure the navigations by hand.

In the overridable method **OnModelCreating**, which we have used for the seeding, you can configure the navigations by hand if needed. You can reach the same result as the other two navigation types but set and configured by hand.

One-to-one relation

Let's start with the easiest relationship; the one-to-one. A movie can have one relationship with one genre. For this to work, we need to create an entity **Genre** and tell the *Movie* it has a relationship with one genre.

The Genre entity

Let's create a new class in the *MovieLibrary.Domain* and call it **Genre**. This new class will only have 2 properties: Id (int) and Name (string). The maximum length of the Name should be 10.

When done, we add the Genre to the **DataContext** with the **DbSet**. This will allow us to create a new migration and update the database. We should also add some seeding:

- Action
- Comedy
- Horror
- Sci-Fi
- Animation

We don't need a service class in the business, because we just need the data, not the logic. Make sure to submit the changes to your database.

Connecting the entities

How does the movie know which genre it needs? The answer is; Adding a reference and a foreign key to the movie.

The foreign key is the primary key of the genre, which is Id. And the reference is the entity Genre. Since we need to add these to the **Movie** entity, we can't add a property Id to it, because it already has an Id. Instead, we use the following code:

```
public int? GenreId { get; set; }  
public Genre? Genre { get; set; }
```

Add these lines to the Movies class, above the ToString() method. The properties Genre and the GenreId are nullable because we don't always want to connect a genre. If the genre needs to be required, we remove the nullable (?).

Entity Framework Core is smart enough to connect the name Genre, which is of the type Genre, to the **GenreId**, which is an int and has "id" in the name. Based on this information, it will create a one-to-one relationship.

Create a new migration and call it CreatingRelationshipMovieAndGenre. Take a look at the code and notice the **UpdateData** method. Since the property **Movie.Genre** is not NULL, it needs to have a value, and Entity Framework Core uses 0 (the default value of an int) to put in the column.

We have learned we can add our own SQL scripts in a migration and this would be a good time to do it.

Remove all the UpdateData-methods and replace them with the following line of code:

```
migrationBuilder.Sql("UPDATE Movies SET GenreId=NULL");
```

Update the database so the relationship is created.

Go to the Program.cs of the MovieLibrary.ConsoleApp and put a breakpoint at line 60 (the line inside the foreach-loop of the method **ShowAllMovies()**). Run the application and make the breakpoint hit. Check out the contents of the movie:

The screenshot shows a debugger window with a list of movie objects. The first object is selected, showing its properties:

Property	Value
Description	"Katriss volunteers to replace her sister in a tournament that
Genre	null
GenreId	null
Id	1
Rating	0
ReleaseDate	{21/03/2012 00:00:00}
Seen	false
Title	"The Hunger Games"

The property Genre is NULL and so is the GenreId. Let's add a genre to the movie The Hunger Games. Go to the data of the table Movies and change the GenreId to the id of the genre action.... I'll wait.

Run the application again. The Genre is still NULL, but the GenreId is the id of the genre action. What happened? Why is the Genre NULL?

Reason: The related data is not queried....

Related data query

There are 3 ways Entity Framework Core can load the related data:

- Eager
- Lazy
- Explicitly

All three have pros and cons and one is not better - or worse - than the other.

Eager loading

If you want to get the related data as soon as you request information from the database, you use eager loading. You explicitly tell Entity Framework Core to include the data. When you use the DataContext to retrieve information you add the **Include()** method to include the object(s) you want to retrieve too.

If we apply this method in the **MovieService**, method **Get()**, it would look like this:

```
public IEnumerable<Movie> Get() => [... DataContext.Movies.Include(x
=> x.Genre)];
```

The Include() is part of the Entity Framework Core library and it adds a left join to the query:

```
SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[Rating], [m].
[ReleaseDate], [m].[Seen], [m].[Title], [g].[Id], [g].[Name]
FROM [Movies] AS [m]
LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id]
```

What happens is that you explicitly say “Give me all the movies, but also the related genres.” And that is translated to a SELECT with a LEFT JOIN to the Genres. A LEFT JOIN joins the tables but returns all the records from the left table and matching rows of the right table. But if there are no matching rows from the right table, the left table row is still showing, but the fields of the right table are NULL.

Lazy loading

If you want to load the related data all the time without you telling it to Entity Framework Core you can configure Entity Framework to load the related data as soon as you want it. This can be done by using a property.

Lazy loading means that the data will be loaded when you need it. To achieve this you need to install a package called [Microsoft.EntityFrameworkCore.Proxies](#). You install this package in the same project as the DataContext (MovieLibrary.Business in our case).

This package gives you a new extension for the optionsBuilder in the OnConfiguring method: **UseLazyLoadingProxies()**. The DataContext part should look like this:

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
    string connectionString = "Data Source=(localdb)\
\MSSQLLocalDB;" +
        "Initial Catalog=MoviesLibrary;" +
        "Integrated Security=True;" +
        "Connect Timeout=30;" +
        "Encrypt=False;" +
        "Trust Server Certificate=False;" +
        "Application Intent=ReadWrite;" +
        "Multi Subnet Failover=False";

    optionsBuilder.UseLazyLoadingProxies().UseSqlServer(connectionString);
}
```

If you want to test this, make sure you remove the Include() from the previous part where we talked about explicit loading.

If all goes “well” you should see an error:

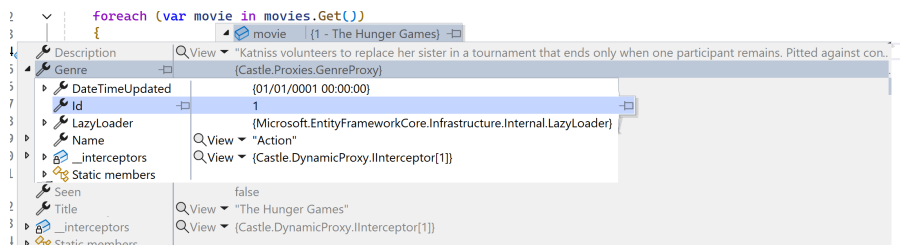
Property 'Movie.Genre' is not virtual. 'UseChangeTrackingProxies' requires all entity types to be public, unsealed, have virtual properties, and have a public or protected constructor. 'UseLazyLoadingProxies' required only the navigation properties be virtual.

The property **Genre** of the entity **Movie** is not virtual. That is correct... But why?

When you make a property virtual, Entity Framework Core creates a special version of your class, called a proxy. This proxy can do extra things, like loading related data from the database when you ask for it.

As we know from the very basics of C# and polymorphism, methods can only be overridden when they are virtual. To let Entity Framework Core create this special version of the class, it needs to override the property.

Let's make the property virtual, run the application again, and check out the contents of the Genre when the breakpoint is hit:



The **Castle.Proxies.GenreProxy** is the proxy, created by Entity Framework Core. You can also see the Id, which is 1, and the name, which is **Action**.

You can't really see it, but as soon as you open the watch of the movie, Entity Framework Core will request the information of the genre, not before.

Explicit Loading

With explicit loading, you load the related data by executing different queries. In the case of the genre, the **Get()** method in the class Movies would look like this:

```
public IEnumerable<Movie> Get()
{
    List<Movie> movies = [... dbContext.Movies];

    foreach (Movie movie in movies)
    {
        movie.Genre = dbContext.Genres.Single(x => x.Id ==
movie.GenreId);
    }

    return movies;
}
```

Yes, this is way more code than the Get is initially.

This isn't the best way because you execute the query to get all the movies and execute single queries for each movie to find the genre.

Explicit loading is a good way if you want to load specific related data for specific records. For example, you only want the genres when the id of the genre of a movie is 2.

Another reason not to use it is database load. If you use explicit loading, the database gets more requests. If you are just you and no other users, there is no problem. If you have a web application that is used by hundreds of users, the database could be really busy.

Especially when you are working with a cloud database. You (usually) pay per transaction. The more transactions you make, the more you pay.

Which one to use?

There is no wrong or right. It all depends on your situation and your needs. I would go for the lazy or the eager loading.

The lazy loading will make sure you don't forget to include a relationship, but with the eager loading, you have more control.

In the situation of this book, I would go for the eager loading. But I wouldn't stop you if you want to go for the lazy loading.

If you want to follow this book to the letter, change the Get() method in the Movies class to the following code:

```
public IEnumerable<Movie> Get () => [... DataContext.Movies.Include(x
=> x.Genre)];
```

One-to-many relation

In the previous chapter, I explained, with a detour on how to load related data, how a one-to-one relation works. Besides one-to-one, there is also a one-to-many, which means one entity has a relationship with multiple entities.

A movie has multiple actors... And there you have a perfect example of a one-to-many relationship.

ICollection

To make sure the movie knows there are actors, we need to add a collection of the type Actor to the Movie class. We could use a

List<T> or an IEnumerable<T>, but we won't.

An **IEnumerable**<> is used for a list of objects that only needs to be iterated through. We use **List**<> for a list of objects that need to be iterated through, modified, sorted, etc. We use **ICollection**<> for a list of objects that need to be iterated through and modified.

If we want to use lazy loading we want to use the virtual ICollection. By default, navigation properties in Entity Framework Core come with change tracking and are proxies. For the dynamic proxy to be created as a navigation property, the virtual type must implement ICollection.

We want to be able to iterate through the list of items but also sort and modify when needed. And since there is a possibility of using lazy loading, we should go for the ICollection.

Let's add a collection of actors to the movie:

```
public ICollection<Actor>? Actors { get; set; }
```

I made it nullable because sometimes a movie does not have actors or the actors are not known yet.

All we have to do now is create a migration and update the database.

Retrieving the data

If you open the data of the table Movies, nothing has changed. But when you open the table Actors, you see a new field: MovieId. Entity Framework Core has created a foreign key from the Actors to the Movies, making sure they can be connected.

Max Rows: 1000				
	Id	Name	DateOfBirth	MovieId
▶	1	Keanu Reeves	02/09/1964 00:...	NULL
	2	Mike Myers	25/05/1963 00:...	NULL
	3	Jim Henson	24/09/1936 00:...	NULL
	4	Jennifer Lawren...	15/08/1990 00:...	NULL
*	NULL	NULL	NULL	NULL

Carrie-Anne Moss also played in The Matrix. Add her to the Actors table.

Next, add all the IDs of the movies in the MovieId behind the actors. For Keanu Reeves and Carrie-Anne Moss use The Matrix.

Let's use the search function of our console application. Go into the

Movies class of the *MovieLibrary.Business* and include the **Actors** property:

```
public IEnumerable<Movie> Get(string query) =>
    [.. dbContext.Movies.Include(x => x.Actors)
     .Where(x => x.Title.Contains(query)
              || x.Description.Contains(query))];
```

Make sure to set a breakpoint on line 76 in the *Program.cs* of the console application. Run the application and search for 'matrix'.

When the breakpoint is hit and you expand the **searchResults**, you will see two actors. Expanding those will show Keanu Reeves and Carrie-Anne Moss.

Join tables

Maybe you already spotted the problem: What if an actor plays in different movies? For example: Keanu Reeves plays in *The Matrix*, which is in the *Movies* table. But he also plays in *John Wick*. How can we reuse the Keanu Reeves entry for both movies? We can't.

To demonstrate, let's add the movie *John Wick* to the *Movies* table. You can find all the information about this movie online.

What we need is a table that connects the actors with the movies. We call these tables *join tables*. They literally join information from different tables. To accomplish this we need a bit more code than the previous one-to-many.

Undo the previous migration if you typed along. Also, remove the **ICollection<Actor>** from the *Movie* class.

Entity for the join table

We start by creating a new entity that will join the *Movie* and the *Actor* table together. It will only contain a foreign key for a movie and one for the actor. It will also contain the navigation element to the entities *Movie* and *Actor*. It does not need anything else, like a service class.

Create a new class in the *Models* folder of the *MovieLibrary.Domain* and call it **MovieActor**. The name already suggests it's an actor for a movie.

```
public class MovieActor
```

```
{  
    public int MovieId { get; set; }  
    public Movie Movie { get; set; }  
  
    public int ActorId { get; set; }  
    public Actor Actor { get; set; }  
}
```

That's it! Nothing more! This entity will be a one-to-one but used multiple times. Makes sense? If not, it will when we have implemented it as a whole.

We also need to add it to the database table. Add the **MovieActor** to the **DataContext** class by using the **DbSet<>**. Make sure to call the property **MovieActors** (plural). Update the database when done.

But creating the migration will result in an error:

Unable to create a 'DbContext' of type ". The exception 'The entity type 'MovieActor' requires a primary key to be defined.

Don't be scared! This is logical. Have you looked at the **MovieActor** class? There is no ID or primary key. And we don't want one. We want to use this entity as a join table.

How to fix this? Answer: **Composite primary key**.

In the **OnModelCreating** method of the **DataContext**, we can configure the composite primary key for the join table and set up the necessary foreign key relationships. This way we tell Entity Framework Core how to handle the data and structure.

We need to tell Entity Framework Core that the properties **MovieId** and **ActorId** of the **MovieActor** can be used together. This combination is the composite primary key.

```
modelBuilder.Entity<MovieActor>()  
    .HasKey(ma => new { ma.MovieId, ma.ActorId });
```

The **HasKey()** is used to define a primary key if it's not logical to Entity Framework Core. This is usually the property **Id** or the property marked with the **[Key]** attribute. But the **MovieActor** doesn't have any of these.

But we are not done yet. We need to tell Entity Framework Core that the entity **MovieActor** has one movie, but many actors. Use the following piece of code and place it below the composite key:

```
modelBuilder.Entity<MovieActor>()
    .HasOne(ma => ma.Movie)
    .WithMany(m => m.MovieActors)
    .HasForeignKey(ma => ma.MovieId);
```

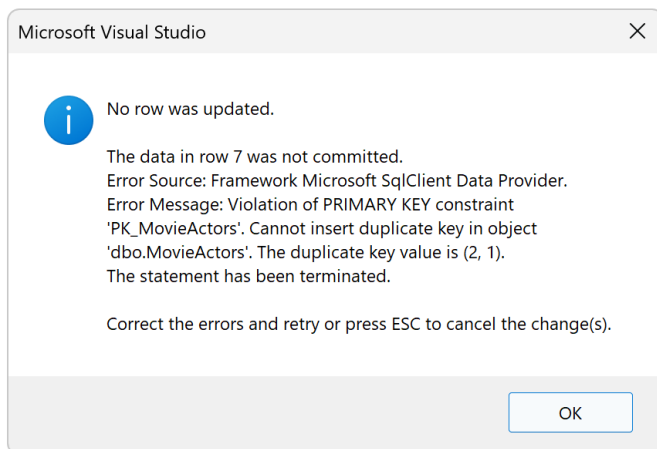
Let's break it down:

First, we select the **MovieActor** with the Entity <>. The **HasOne** specifies that each **MovieActor** has one **Movie**. The **WithMany** tells Entity Framework Core that one **Movie** can have many **MovieActors**. The **HasForeignKey** tells Entity Framework Core that the **MovieActor** entity has a foreign key called **MovieId**.

All together this is what we want: One movie with multiple actors.

You can now update the database.

Another big advantage of this kind of table, especially with the composite primary key, is that the combination of the **MovieId** and the **ActorId** has to be unique. If you have a **MovieId** of 1 and an **ActorId** of 2, you can't enter it again. It will show you an error:



Add navigation property to Movie

Add navigation property to Movie

We want to have the actors for a movie, not the movies for an actor. Although we can make it both ways (a many-to-many), we want only the actors for a movie.

With the first attempt to create a one-to-many, we used an **ICollection<T>** and we will again. In the **Movie** class, add the

navigation property to the MovieActor like this:

```
public ICollection<MovieActor>? MovieActors { get; set; }
```

This will be the first step to retrieve the data, but we are not there yet. We need to tell Entity Framework Core to load the data with the query. Since 'we' decided to use eager loading, we need to use the **Include** again.

Including

But we need to include two entities. First the MovieActor and, from MovieActor, the Actor. We can use the Include multiple times, but (... and there is always a but) the Include only works on the current entity. The following code won't work:

```
public IEnumerable<Movie> Get(string query) =>
[.. dbContext.Movies
    .Include(x => x.MovieActors)
    .Include(x => x.actors)
    .Where(x => x.Title.Contains(query)
        || x.Description.Contains(query))];
```

The **x.actors** will give an error because the first Include() is set on the Movies and the second Include() is set on the same Movies; the Movies entity doesn't have a property actors, the MovieActors does.

To include an entity inside an included entity we use the **ThenInclude()**. This one is used for eager loading related data in a nested manner, like the Movies -> MovieActors -> Actors. It allows us to load related entities that are further related to the entities we are already loading.

```
public IEnumerable<Movie> Get(string query) =>
[.. dbContext.Movies
    .Include(x => x.MovieActors)
    .ThenInclude(x => x.actors)
    .Where(x => x.Title.Contains(query)
        || x.Description.Contains(query))];
```

And this will do the trick. You can go deeper into including entities. Just keep using the **ThenInclude()** method. If you want to go back a step, to a parent entity, use the **Include()** again.

If we want to make this piece code correct, we should also include the Genre. This is the complete code for searching movies:

```
public IEnumerable<Movie> Get(string query) =>
[...dbContext.Movies
```

```
.Include(x => x.MovieActors)
    .ThenInclude(x => x.Actors)
.Include(x => x.Genre)
.Where(x => x.Title.Contains(query)
    || x.Description.Contains(query));
```

Final words

Relationships between tables, or entities, are pretty common. They help us reuse data and connect them with joins. While using Entity Framework Core you can create relationships in different ways.

One-to-one means one entity has one relationship with another entity. We connect them through **navigations** and **conventions**. Making the references property **virtual** and adding the naming contention nameId property to the entity, Entity Framework Core will figure out they are connected.

We have learned there are 3 types of loading related data: **Eager loading**, **explicit loading**, and **lazy loading**. While all have pros and cons, the eager loading is, for now, the best way to go. By using the **Include()** while retrieving the data, we can include the referenced entities and get the related data. Entity Framework Core will translate this to an SQL query with a join.

Besides one-to-one, we also learned about the **one-to-many**. One entity has a relationship with one or more entities. For this to work we need to use **ICollection<T>**, add a join table, create a **composite primary key**, and put it all together by loading the related data. if we want to include an entity in an included entity, we use the **ThenInclude()**.

Chapter 8: Stored procedures

In the very first chapter, we looked at stored procedures. We now know what they are, how we can create them, and how to execute them to get results.

Entity Framework Core cannot create stored procedures, but we can execute them and grab the results in a C# class.

In this chapter

In the chapter *Let me introduce you to the database* we talked about stored procedures and how they work with SQL. If you want to know more about stored procedures, scroll back a bit.

Entity Framework Core also has a way of executing stored procedures, retrieving the information, and allowing C# to work with it.

In this chapter, we will create two stored procedures and learn how Entity Framework Core can execute them. One stored procedure has a parameter and the other one does not. The parameter is used to learn the **SqlParameter** object, which is important for several reasons.

A stored procedure is not an entity and we need to execute it by a string query. We will use the **FromSqlRaw()** method on the **DataContext** to execute such a query. One stored procedure returns a result that cannot be linked to any entities we have and the other can be linked to the **Movie** entity. This means we can use two different approaches to catch the data from the stored procedures.

Entity Framework Core can only execute stored procedures, not create them.

A simple stored procedure

In a previous chapter *Migrations* we have created a simple stored procedure called **GetMovies**. Let's learn how we can use Entity Framework Core to execute that stored procedure and retrieve the data from it.

An entity

Like most things you want to do with retrieving data, you need an entity. An entity is the very basic of Entity Framework Core; it represents the object with the types you expect to retrieve from the database table.

Stored procedures are in SQL a special function, but for Entity Framework Core is just another element that returns data, like a database table. And we can handle it as such.

That is why we will start with a new entity in our solution. Let's create a new class in *MovieLibrary.Domain*, folder Models. The stored procedure is called **GetMovies**. To give the new class the same name doesn't really work. Let's call the class *MovieClassification*.

The stored procedure exposes 3 columns: Title, Seen, and Classification. All are strings, so that makes it pretty easy. The entity *MovieClassification* should have the same fields but as properties;

```
public class MovieClassification
{
    public string Title { get; set; }
    public string Classification { get; set; }
    public string Seen { get; set; }
}
```

With the entity created, we can now grab the data from the stored procedure and store it in the *MovieClassification*.

Execute the stored procedure

We don't add the *MovieClassification* class to the *DbSet*. If we did, Entity Framework Core would create a new table with the information from the class. We don't want that. Instead, we need to execute a piece of SQL inside C# code. Let's add it to the **Movies** class inside *MovieLibrary.Business*.

Entity Framework Core has a way to let you set the model you want to use (*MovieClassification*) that will be used to parse the information from the database to C#. For this, we use the **Set()** method on the *DataContext*.

When the model is set, we can execute a raw query string. Meaning that we let Entity Framework Core execute **exec GetMovies**.

And that's it! In code, it looks like this:

```
public IEnumerable<MovieClassification> GetClassifications()
{
    return [.. dbContext
        .Set<MovieClassification>()
        .FromSqlRaw("EXEC GetMovies")]
}
```

Don't forget to add the method to the interface!

To see some results of the data from the stored procedure, let's add a new endpoint to the API. Since this is an e-book about Entity Framework Core, I won't explain much about the API endpoint:

```
app.MapGet("/api/movies/classifications", (IMovies movieService) =>
{
    return movieService.GetClassifications();
})
.WithName("GetClassifications")
.WithOpenApi();
```

We can now run the API and see the result.

Code Details

500
Undocumented Error: response status is 500

Response body

```
System.InvalidOperationException: Cannot create a DbSet for 'MovieClassification' because this type is not included in the model for the context.
   at Microsoft.EntityFrameworkCore.Internal.InternalDbSet`1.get_EntityType()
   at Microsoft.EntityFrameworkCore.Internal.InternalDbSet`1.CheckState()
   at Microsoft.EntityFrameworkCore.Internal.InternalDbSet`1.get_EntityQueryable()
   at Microsoft.EntityFrameworkCore.Internal.InternalDbSet`1.System.Linq.IQueryable.get_Provider()
   at Microsoft.EntityFrameworkCore.RelationalQueryableExtensions.FromSqlRaw[TEntity](DbSet`1 source, String sql, Object[] parameters)
   at MovieLibrary.Business.Movies.GetClassifications() in C:\Users\kenji\source\repos\EntityFrameworkEBook\MovieLibrary\Business\Movies.cs:line 50
   at Program.<c.<<Main>>>b__0_3(IMovies movieService) in C:\Users\kenji\source\repos\EntityFrameworkEBook\MovieLibrary\API\Program.cs:line 56
   at lambda_method10(Closure, Object, HttpContext)
   at Microsoft.AspNetCore.HttpPolicy.HttpsRedirectionMiddleware.Invoke(HttpContext context)
   at Microsoft.AspNetCore.StaticFiles.StaticFileMiddleware.Invoke(HttpContext context)
   at Swashbuckle.AspNetCore.SwaggerUI.SwaggerUIMiddleware.Invoke(HttpContext httpContext)
   at Swashbuckle.AspNetCore.Swagger.SwaggerMiddleware.Invoke(HttpContext httpContext, ISwaggerProvider swaggerProvider)
   at Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddlewareImpl.Invoke(HttpContext context)
```

HEADERS

```
=====
Accept: application/json
Host: localhost:7182
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/126.0.0.0 Safari/537.36
Accept-Encoding: gzip, deflate, br, zstd
```

Download

Okay, that didn't go as planned. But why?

The error tells us that the `MovieClassification` is not included in the model for the context. The context is `DataContext` and we didn't add the `MovieClassification` to it through `DbSet`, because we don't want to create a table from the model.

Notice I call it a model and not an entity? This is because we haven't made it an entity and Entity Framework Core has no idea how to use

it since it's not known.

The ModelBuild

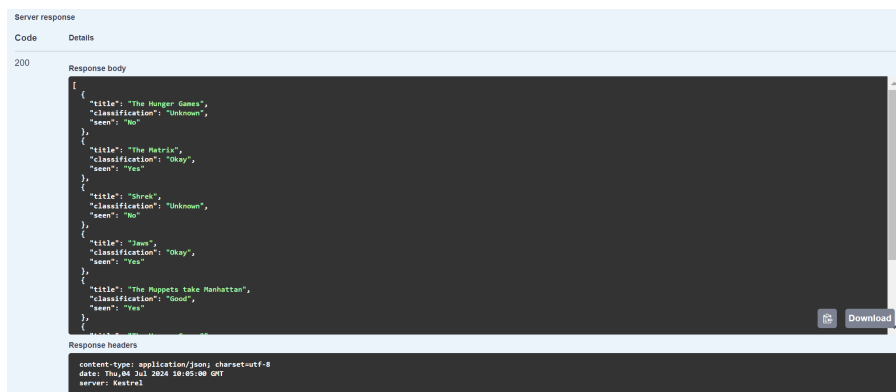
To add the model to Entity Framework Core, and make it an entity, we need to add it through the **ModelBuilder**, which is located in the **OnModelCreating**. The way we add it to Entity Framework Core is with the Fluent API. You have used it before (seeding and relationships). We go more into detail on the Fluent API in another chapter.

We need to connect the **MovieClassification** model to the stored procedure **GetMovies** (or the other one around, you choose). However, all the entities need a primary key for identification and uniqueness. Lucky for us, we can remove that requirement through the Fluent API for this specific entity.

All you have to do is point to the entity, remove the primary key requirement, and point to the stored procedure:

```
modelBuilder.Entity<MovieClassification>()  
    .HasNoKey()  
    .ToView("GetMovies");
```

Run the API again and execute the correct endpoint and you will see a nice result, just how we want it.



Stored procedure with parameters

Calling a stored procedure is pretty easy, as shown in the previous section. But a stored procedure can also contain parameters, much like a method in C#.

To demonstrate this, let's create a second stored procedure that takes a parameter:

```
CREATE PROCEDURE Search
    @query varchar(200)
AS
    SELECT
        [m].[Id], [m].[Plot], [m].[GenreId], [m].[IsDeleted], [m].
        [LastModifiedDate], [m].[Rating],
        [m].[ReleaseDate], [m].[Seen], [m].[Title], [g].[Id], [t].
        [MovieId], [t].[ActorId], [t].[Id],
        [t].[DateOfBirth], [t].[Name], [g].[Name]
    FROM [Movies] AS [m]
    LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id]
    LEFT JOIN (
        SELECT [m0].[MovieId], [m0].[ActorId], [a].[Id], [a].
        [DateOfBirth], [a].[Name]
        FROM [MovieActors] AS [m0]
        INNER JOIN [Actors] AS [a] ON [m0].[ActorId] = [a].[Id]
    ) AS [t] ON [m].[Id] = [t].[MovieId]
    WHERE [m].[Title] LIKE '%' + @query + '%' OR [m].[Plot] LIKE
    '%' + @query + '%'
    ORDER BY [m].[Id], [g].[Id], [t].[MovieId], [t].[ActorId]
```

This stored procedure receives on parameter: query. The result of this stored procedure is all the movies that contain that query in the title. Nothing fancy, I just copied it from the Entity Framework Core log when I search for a movie.

Adding to C# and Entity Framework Core

We don't need a special model for this, because the stored procedure will send a list of movies back to our application, just like a simple get with Entity Framework Core.

All we need to do is replace the current code of the **Get(string query)** method in the **Movies** class with code that makes Entity Framework Core request the movies through the stored procedure.

Since this entity and the result have a primary key, we don't have to tell Entity Framework Core not to use a key. We can just execute the stored procedure. And because of that, we can use the Movies entity from the DataContext.

Because it has a parameter we could just execute the stored procedure like this:

```
public IEnumerable<Movie> Get(string query)
{
    return dbContext.Movies.FromSqlRaw("@EXEC Search
{query}").ToList();
```

```
}
```

But that is not really a good way to use parameters in SQL. It's better to use the SqlParameter object. And that for a few reasons:

- Prevent SQL Injection
SQL injection is a real thing when it comes to queries that are built up with strings and user input. SqlParameter prevents this.
- Type safety
Using SqlParameter you can ensure that the data types of the parameters match the expected data types in the stored procedure. This reduces errors during runtime.
- Better performance
Using SqlParameter you can ensure that the data types of the parameters match the expected data types in the stored procedure. This reduces errors during runtime.
- Clarity
The code needs to be readable and clear for everyone who looks at it. SqlParameter can help with that when it comes to parameters for stored procedures.

This means we need to initialize a SqlParameter and use it when we call the stored procedure:

```
public IEnumerable<Movie> Get(string query)
{
    SqlParameter queryParam = new SqlParameter("@query", query);
    return dataContext.Movies.FromSqlRaw("EXEC Search @query",
    queryParam).ToList();
}
```

When you run the API or the console application you will notice nothing has changed on the outside. When you search for a movie, the same result is shown.

Final words

The question that we have now is; what is better: using the entities or a stored procedure? And as you might already expect; it depends on the situation.

If you want full control over the database, have a very complex query, and want to have the maximum performance, are working on a legacy system that already has stored procedures, and want to use the database security system: Use stored procedures.

If you want to keep the developer in control, use LINQ to keep the logic in the business area of the application, and able to easily refactor the code if there is a change: Use Entities.

It's not really an 'or' situation, because one situation leads to the next. It's not a great idea to use a stored procedure, call it from C#, and use extensive LINQ on that call in C#.

Like with most situations when developing software you need to know what you are working with and what the situation is. Always keep an eye on the performance, readability, and maintainability.

Chapter 9: API Implementation

In the type-along solution, you also find a minimal API. A minimal API with ASP.NET Core basically removes the Startup.cs and places that code in Program.cs. Controllers, with the actions, are removed and placed in the Program.cs as mappings. It allows us to keep the coding for the API at a bare minimum.

Using Entity Framework with an API could work the same as with a console application. But I want to use dependency injection. Using this could break the console application, but we will fix that in a moment.

Another reason to use the API is to see more of Entity Framework at work.

In this chapter

We have been using the console application for a while now. Time to look at the API. Although most of the subjects can be used inside the console application, I believe the API makes it a bit easier to see and feel what is happening.

We let go of the hardcoded initialization of the DataContext and use **dependency injection** to initialize and set up the DataContext, and also DbContext. The **connection string** will be placed in a more general and secure location called the **app settings**.

By using the **DbContextOptionsBuilder** we can tell Entity Framework which data server type to use and set up much more. Other options of the DbContextOptionsBuilder are spread through the upcoming chapters.

When the setup for the dependency injection for the **DataContext** is complete we will be **injecting** the DataContext in the necessary locations and remove the hardcoded initialization.

The way we handle **migrations** will be a bit different and I will tell you why and how to proceed with the migrations.

After the API is set up, the console application is broken. We need to fix this before we continue. The idea is that both applications use the same database. That means we need to add the DataContext to the

services set up of the console application.

But now the console application shows something like **INFO**. What is this and, more importantly, how do we get rid of it?

Why use dependency injection?

Again, good question! There are several reasons why we use dependency injection.

Dependency injection is a way to initialize the service just once and use that initialized version all the time. It initializes the services in one place and then you provide them to the class where you need them.

A problem, or more an architectural error, is that when we want to switch to something else than MSSQL, Oracle for example, we need to change a lot of code in the DataContext. Instead, we could create a new DataContext, build it specially for Oracle, and “flip a switch” with dependency injection to not use MSSQL but Oracle.

You can see dependency injection as a way to manage the connections between different parts of your code by letting you inject the components you need into a specific class. That means you don't have to worry about the fact if the service you have injected and want to use, is created or not.

Dependency injection makes your code easier to read, maintain, and test.

Setting up dependency injection with Entity Framework Core works a little bit differently than with normal classes.

Storing the connection string

Let's first take a look at the connection string. This one is still hardcoded and should be configurable. Time to move it to the app settings.

The app settings should already be known to you, but if not: The app settings is a JSON file that can contain configurable information and can be set up for different environments. The MovieLibrary.API already has app settings; appsettings.json

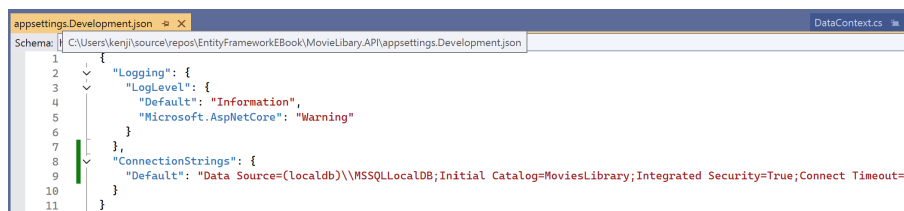
It also has a nested file called `appsettings.Development.json`. This one is specially made for environment Development, which is your local machine. You can also add other environments. Other popular environments are Accept, Test, and Production.

A new section

Each environment has the same keys, but different values. A connection string is an example of the same key, but different values. Your local database can't be reached by the production environment and the other way around.

Let's start by opening the `appsettings.Development.json`. We are going to create a new section called **ConnectionStrings**. This section is predefined and is known by .NET.

This section gets a key called **Default**. The default connection string will be stored here. You can add others if needed, but let's stick with the default first. Use the hardcoded connection string from the `DataContext` as the value:



```
appsettings.Development.json
Schema: C:\Users\kenji\source\repos\EntityFrameworkEBook\MovieLibrary.API\appsettings.Development.json
1 {
2   "Logging": {
3     "LogLevel": {
4       "Default": "Information",
5       "Microsoft.AspNetCore": "Warning"
6     }
7   },
8   "ConnectionStrings": {
9     "Default": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=MoviesLibrary;Integrated Security=True;Connect Timeout=
10  }
11 }
```

You could copy the section `ConnectionStrings` and paste it into a different app settings for a different environment (`appsettings.Production.json` for example) and change the connection string in the `Default`. When you publish the API to a production environment, the connection string from the `appsettings.Production.json` is used, not the `appsettings.Development.json`.

Retrieving the connection string

After this, you can remove the method **OnConfiguring** from the **DataContext** class.

Now we need to retrieve that connection string from the app settings and store it in a variable. We then will use that variable, with the connection string, to initialize Entity Framework Core.

Add the following code after the line **var builder = ...** in the *Program.cs* of the API:

```
string connectionString =  
builder.Configuration.GetConnectionString("Default");
```

This will search the app settings for the section **ConnectionStrings** and the key **Default**. If one or both are not found, an error will occur.

By placing the connection string inside the app settings we have centralized the connection string, which makes it easier to change (if needed). But we also made it a bit safer.

And because we can work with environments, we should also create an *appsettings.Production.json*. When we publish the API to production, the API should use the connection string for the production environment, not the development environment.

To demonstrate this, create a second app settings file and call this *appsettings.Production.json*. Copy and paste the data from the development settings into the production file. Change the **Initial Catalog** of the connection string in the production file to **MoviesLibrary_Production**.

We will come back to this later.

Adding DataContect to services

Like all objects in dependency injection, we need to add it to the services. But not with the `builder.Services.AddScoped<>`, or something like that. Entity Framework Core has its own method for that: **AddDbContext**.

AddDbContext

The `AddDbContext` will add the `DbContext` to the services. This extension method also has a few parameters: `DbContextOptionsBuilder` and the `ServiceLifetime`.

The `DbContextOptionsBuilder` should look familiar because it's the same one we have used in the `DataContext` when overriding the `OnConfiguring`. The parameter of this method is the same `DbContextOptionsBuilder`.

The `ServiceLifetime` is the is the lifecycle of the service we want to

configure. We have three flavors: scoped, transient, and singleton.

Let's start with the first part of registering the DbContext:

```
string connectionString =  
builder.Configuration.GetConnectionString("Default");  
builder.Services.AddDbContext<DataContext>();
```

We can't use DbContext itself, because that's a class that has no idea of our entities and we can't touch it. But we have the DataContext, which inherits from DbContext. Hence the reason we configure the DataContext for the dependency injection.

We now have to tell Entity Framework Core which type of database we want to use. This works the same as in the OnConfiguring, but inside the first parameter of the AddDbContext method:

```
string connectionString =  
builder.Configuration.GetConnectionString("Default");  
builder.Services.AddDbContext<DataContext>(  
    options=> options.UseSqlServer(connectionString)  
);
```

The first parameter is an expression and it can be used to set up the Entity Framework Core connection. At first, we want to set up the connection string, otherwise Entity Framework Core has no idea where to find the database.

The options can also be used to set up logging and much more. We will come back to that later.

The second parameter is the lifecycle, which each configuration of the dependency injection has. By default, the DataContext will have a lifecycle of **scoped**.

Scoped will create a new instance for the class (DataContext) with each new HTTP request. This means that all the classes, that get this dependency injected, will get the same instance. When the request is finished, the instance is disposed of.

Useful when you want each request to use the same instance, but different requests use different instances.

And this is also the best lifecycle we can use. The reasons are:

- Using a scoped lifecycle ensures that each HTTP request gets its own instance of DataContext. This allows for changes to be tracked and saved correctly within the request's scope.

- If you share the DataContext you have potential concurrency issues. You don't want to share the DataContext with other requests.
- Creating a new DataContext for each request is better because it avoids the overhead of creating a new instance for every operation, like with the transient lifecycle.

Although the default lifecycle is scoped, I am a big fan of placing it in the code explicitly. This way you know it's there for a reason. And maybe you want to change the lifecycle, with a good reason. This is how you set the lifecycle for the DataContext :

```
string connectionString =  
builder.Configuration.GetConnectionString("Default");  
builder.Services.AddDbContext<DataContext>(  
    options => options.UseSqlServer(connectionString),  
    ServiceLifetime.Scoped  
);
```

But we are not done yet. How does the DataContext class know about the settings? All we have done is set it up, but we haven't sent any of the information - like the connection string - to the DbContext. The DbContext isn't even used in the configuration.

A DataContext constructor

The AddDbContext is used to set up the dependency injection and create an instance of it. We can use this instance to inject it into the classes, where needed. We only configured it but did not use that configuration.

If you run the API you will get this error:

'AddDbContext' was called with configuration, but the context type 'DataContext' only declares a parameterless constructor. This means that the configuration passed to 'AddDbContext' will never be used. If configuration is passed to 'AddDbContext', then 'DataContext' should declare a constructor that accepts a DbContextOptions < DataContext > and must pass it to the base constructor for DbContext.

That "parameterless constructor" is a hint. The options need to be sent to the constructor of the DataContext. Then we need to send those options to the DbContext. Only then we have created a correct configuration for dependency injection and Entity Framework Core.

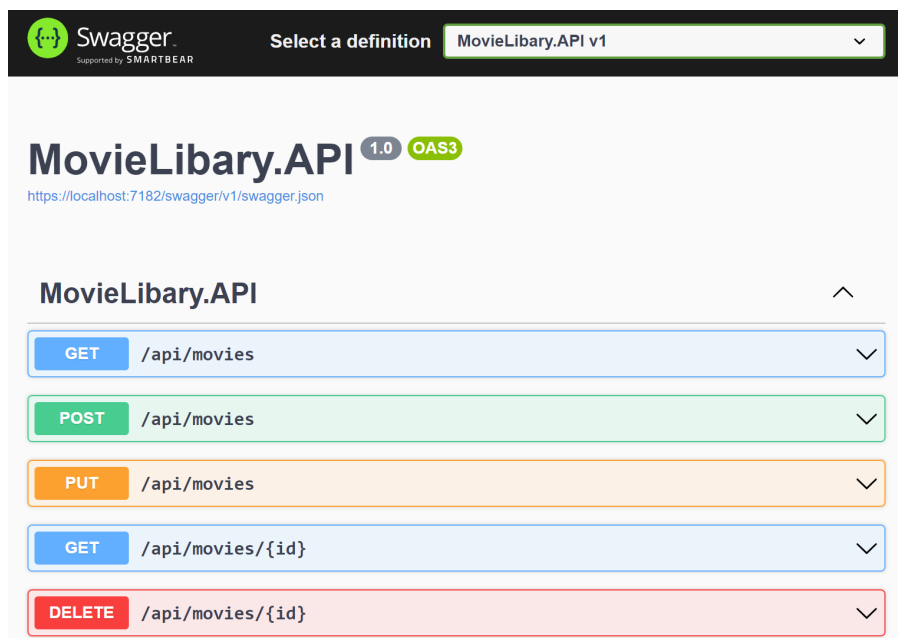
Create the following constructor in the **DataContext** class:

```
public DataContext(DbContextOptions<DataContext> options) :  
    base(options) { }
```

When the `DataContext` is set up for the dependency injection and the options, the `DataContext` is initialized and the options from the API are sent to the `DataContext`. The `DataContext` constructor will send the options to the `DbContext`, which has a constructor for the options.

You can also use the primary constructor on the `DataContext` if you want. When you do you will see that the **`DbContext`** is used instead of **`base`**.

If you start the API now you will see Swagger.



Using the injection

All is set now and the only thing that remains is using the injection of the `DataContext`. At this point, we have two locations where we use the `DataContext`: **`Actors`** and **`Movies`**. Let's start with the **`Movies`** class.

Injecting into Movies

Injecting a service is pretty easy; create a constructor, if there is none, and use the interface or class name that is configured for injection.

The DataContext isn't any different. This is the constructor for the Movies class:

```
public Movies(DataContext dataContext)
{
    this.dataContext = dataContext;
}
```

But since we are working in .NET 8 and C#12, let's use the primary constructor:

```
public class Movies(DataContext dataContext) : IMovies
{
    ...
}
```

And that's it! We have now successfully configured dependency injection for Entity Framework Core and injected the DataContext into a class! Awesome!

Your turn! Inject the DataContext inside the Actors class. Make sure this class uses the injected DataContext.

Start the API and you will see Swagger. Try it out and get all the movies.

The screenshot displays the Swagger UI interface. At the top, the 'Responses' tab is selected. Below it, the 'Curl' section shows a command to perform a GET request to the endpoint `https://localhost:7182/api/movies` with the header `accept: application/json`. The 'Request URL' section shows the same endpoint. The 'Server response' section shows a status code of 200. The 'Response body' section displays a JSON array of movie objects. The first object is for 'The Hunger Games' (id: 1, releaseDate: 2012-03-21T00:00:00, rating: 0, description: 'Katniss volunteers to replace her sister in a tournament that ends only when one participant remains. Pitted against contestants who have trained for this all their life, she has little to rely on.', genre: 'Action', movieActors: null). The second object is for 'The Matrix' (id: 2, releaseDate: 1999-03-31T00:00:00).

```
[
  {
    "id": 1,
    "title": "The Hunger Games",
    "releaseDate": "2012-03-21T00:00:00",
    "seen": false,
    "rating": 0,
    "description": "Katniss volunteers to replace her sister in a tournament that ends only when one participant remains. Pitted against contestants who have trained for this all their life, she has little to rely on.",
    "genreId": 1,
    "genre": {
      "id": 1,
      "name": "Action"
    },
    "movieActors": null
  },
  {
    "id": 2,
    "title": "The Matrix",
    "releaseDate": "1999-03-31T00:00:00"
```

Migrations

I know we already had a fairly big chapter about migrations, but things have changed now and migrations as we know it won't work anymore.

The connection string is now moved from the DataContext to the app settings. Or better yet; the connection string is moved to a front-end application.

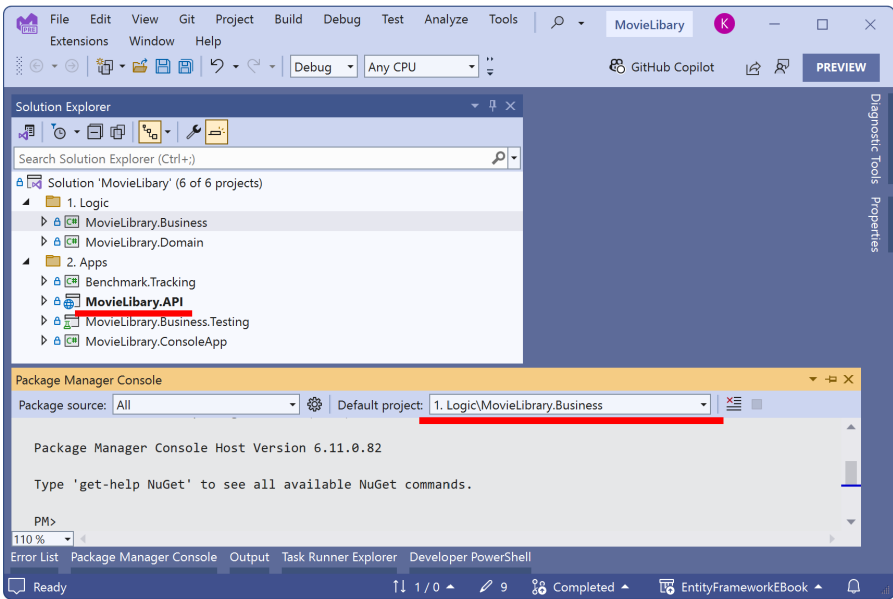
No database provider

If you try to update the database, even though there are no migrations, you will get an error. Make sure you have set the MovieLibrary.Business as the startup project and the default project in the Package Manager Console.

Unable to create a 'DbContext' of type ". The exception 'No database provider has been configured for this DbContext

The original error is much longer, but I think this piece already gives a good idea of what is happening here; Entity Framework can't find the database... Duh! We removed the connection string.

The fix is really easy: Set the API as the startup project and make sure the default project in the Package Manager Console is set to the project that has the migrations and the DataContext.



Let's try it again.

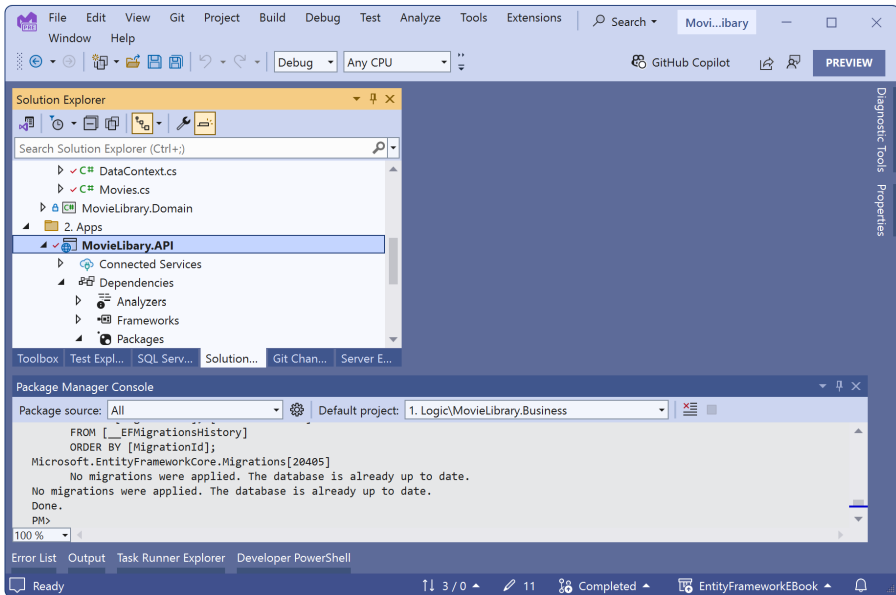
Missing package

Your startup project 'MovieLibrary.API' doesn't reference Microsoft.EntityFrameworkCore.Design. This package is required for the Entity Framework Core Tools to work.

Sigh... Now what? The *MovieLibrary.Business* already has the Entity Framework Core tools packages, which makes sure the commands like `update-database` work. But you can't use that in the API. But the connection string is here... How?

By installing the *Microsoft.EntityFrameworkCore.Design* packages. This package makes sure the *Microsoft.EntityFrameworkCore.Tools* commands can be shared across other projects, like the *MovieLibrary.Business* and the *MovieLibrary.API*. Install the *Microsoft.EntityFrameworkCore.Design* into the *MovieLibrary.API* and try again.

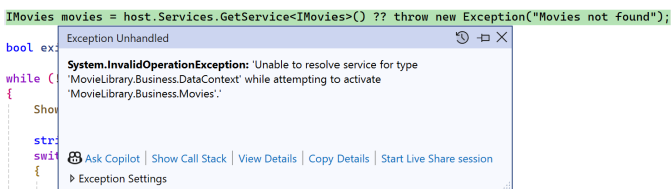
It will now show that there are no migrations were applied.



Future database updates and migrations run through the API, so make sure the API is set as start-up and the *MovieLibrary.Business* is set to the default project.

Fixing the console app

The API works as it should, but the console app will give errors when you try to run it.



The console app is prepared for dependency injection, so the fix is pretty easy; Add the configuration for the DataContext and it will work. It will take a bit more effort to make the console app work.

I am not going into much detail about how this all works. This is a book about Entity Framework Core and not console apps. But it will give you some leverage on looking online for more information.

Getting the configuration

First step: Copy the appsettings.json and the appsettings.Development.json from the API to the console app. The connection string stays the same.

Second step: Install a bunch of NuGet packages... Again.

A console application is the emptiest type of project you can think of. While the API is a project with a lot of packages and libraries installed, the console app has almost none.

Install the following packages in the console application:

- Microsoft.Extensions.Configuration
- Microsoft.Extensions.Configuration.Json
- Microsoft.Extensions.Configuration.EnvironmentVariables

Each package has the purpose of reading the appsettings (Development) and accessing the .NETs configuration functionality.

We will add the following code at the very beginning of the *Program.cs*, under the using:

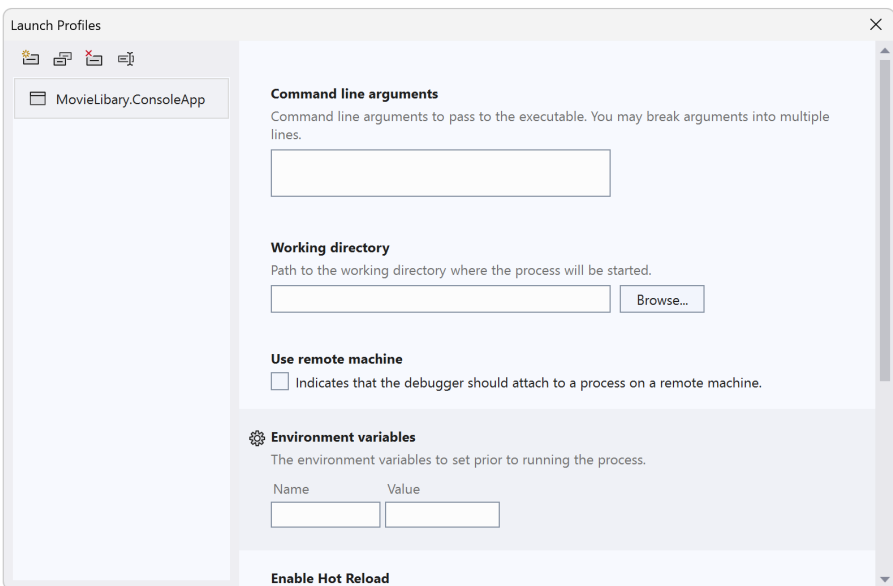
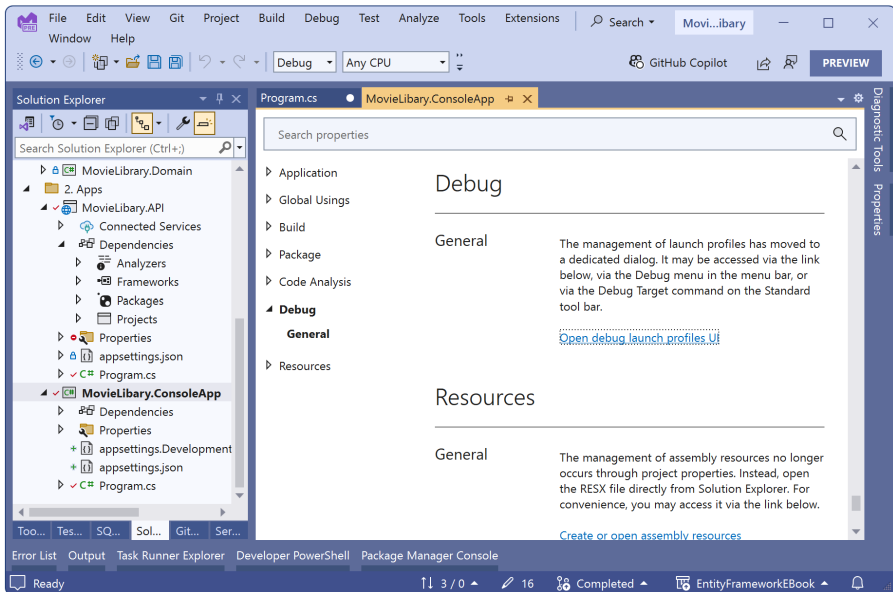
```
IConfigurationRoot config = new ConfigurationBuilder()  
    .SetBasePath(AppContext.BaseDirectory)  
    .AddJsonFile("appsettings.json", optional: true,  
reloadOnChange: true)  
    .AddJsonFile($"appsettings.  
{Environment.GetEnvironmentVariable("ENVIRONMENT")}.json",  
        optional: true)  
    .AddEnvironmentVariables()  
    .Build();
```

This is the code that will load the JSON files appsettings.json and appsettings.ENVIRONMENT.json into the memory of the application.

Setting the environment

The ENVIRONMENT is an environment setting that holds which environment the application is running. The API already has that setting, but the console has not. We need to set it.

Open the properties of the console application. And go to the category Debug. Click on the link “Open debug launch profiles UI”.



Inside the section **Environment Variables**, we can add the variable we need. Type **ENVIRONMENT** with the value **Development**. Close the dialog. (yes, there is no save button)

Get the connection string from the appsettings.Development.json the same way as inside the API and place the following piece of code under the configuration part:

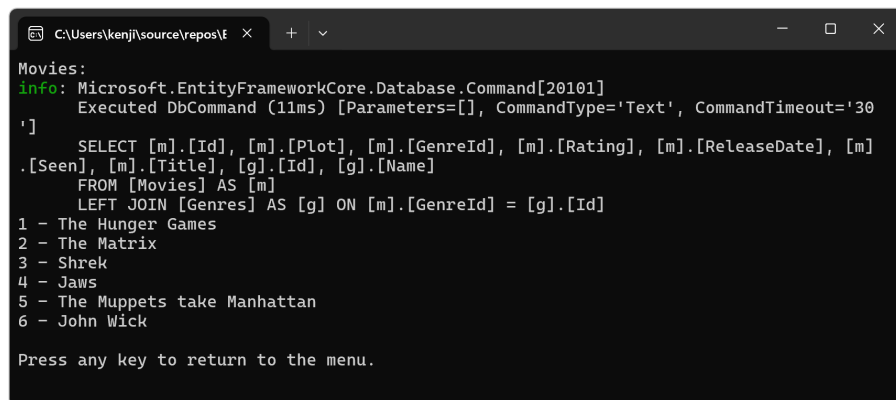
```
string connectionString = config.GetConnectionString("Default");
```

All that remains is getting the `DataContext` inside the services so the `Movies` class will receive an instance of said `DataContext` when we start the application.

This piece is also the same as the API, but inside the **host**, which should now be under the **connectionString** variable:

```
using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddScoped<IMovies, Movies>();
        services.AddDbContext<DataContext>(
            options => options.UseSqlServer(connectionString),
            ServiceLifetime.Scoped
        );
    })
    .Build();
```

The console application will now run and the movies will be shown when you choose option 1.

A screenshot of a Windows console application window. The title bar shows the file path 'C:\Users\kenj\source\repos\...' and standard window controls. The console output starts with 'Movies:' followed by an 'info' message from Microsoft.EntityFrameworkCore.Database.Command[20101] indicating a successful database command execution. Below this, a SQL query is displayed: 'SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[Rating], [m].[ReleaseDate], [m].[Seen], [m].[Title], [g].[Id], [g].[Name] FROM [Movies] AS [m] LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id]'. The results are a list of 6 movies: '1 - The Hunger Games', '2 - The Matrix', '3 - Shrek', '4 - Jaws', '5 - The Muppets take Manhattan', and '6 - John Wick'. The prompt 'Press any key to return to the menu.' is at the bottom.

```
C:\Users\kenj\source\repos\... x + v
Movies:
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (11ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
      SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[Rating], [m].[ReleaseDate], [m]
      .[Seen], [m].[Title], [g].[Id], [g].[Name]
      FROM [Movies] AS [m]
      LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id]
1 - The Hunger Games
2 - The Matrix
3 - Shrek
4 - Jaws
5 - The Muppets take Manhattan
6 - John Wick
Press any key to return to the menu.
```

But what is this info?... Let's talk about logging.

Final words

Dependency injection is a common practice when it comes to C# and .NET. You can easily set up Entity Framework Core to be injected into other classes through the `DataContext`. All you need to do is make sure the **connection string** is in a location that is easy to reach, like the **app settings**. Then you use the **AddDbContext** to put it all together inside the API. Make sure to use the correct lifecycle. By default, the lifecycle is set to **scoped**, but you can also use **transient** or **singleton**.

Then you need to tell the `DbContext` what the settings, like the

connection string, are. You do this by creating a **constructor** that receives one parameter: **DbContextOptions<DataContext>**. The value of this parameter is sent to the DbContexts constructor.

When the API is ready, you can **inject** the DataContext into the classes you want.

The way you create and use the migrations changes a bit since we have moved the connection string from the DataContext class to the appsettings. This means you need to set the **API** to the **startup project** and in the Package Manager Console you need to set the MovieLibrary.Business as the **default project**.

Then to execute the commands, like add-migration and update-database, you will need to **install the package Microsoft.EntityFrameworkCore.Design** into the API project.

To make the **console application** work, you don't have to do much. It has the same principle as the API, but reading the appsettings is a bit different. For this to work, you need a **few packages** and a bit more code to get the ConnectionStrings sections. However, the configuration of the DataContext, the dependency injection of said DataContext, and other settings are the same.

Chapter 10: Logging

Entity Framework Core has its own logging, which you can use or ignore. In this chapter, we will discover different ways of logging the information from Entity Framework Core and how we can investigate what Entity Framework Core is doing.

We will explore easy ways and the most in-depth logging that is possible with Entity Framework Core. This somewhat inspects the queries with query inspectors, but that gets its own chapter.

Be careful when you log the information. Extensive logging can reduce the performance of your application. Use extensive logging while debugging and less logging (or none) when the application is released.

In this chapter

Logging is really important because it can show how your application works and with what kind of data. When a user runs into a problem, logging can usually show what is going wrong.

Entity Framework Core has also the capability to log information. Such as queries, actions, errors, and more. This we already know a bit, but the way we can process that logging is a different story.

In this chapter, we will look at two different ways you can log information and investigate the logging information from Entity Framework Core. We will look at **simple logging** and **Microsoft.Extensions.Logging**. But there are more ways, but these two give the most understandable examples for this subject.

Both ways can **log to a console** or **debug**. If you look deeper into logging, you will find you can also log to files, databases, or the Windows event viewer. But this is not on topic with the Entity Framework Core and pretty easy to find how to do this. Knowing how to start logging in with Entity Framework Core is half of the work.

We will also look at how you can log **sensitive data**, which is turned off by default, and what **log levels** are and how to change them.

The simple way

Let's check out the simplest way of logging in with Entity Framework Core. There is a way to obtain the log information with minimal configuration. This is not much work, but it is harder to control. But if you want to quickly check out what Entity Framework Core is doing, you can use this method.

The simple logging is obtained through the **DbContextOptionsBuilder**. You can find this one in the **OnConfiguring** in the **DataContext** class or when you set up Entity Framework Core in the services of your application.

Since we have our **DataContext** configured with the API services, we use that one and it currently looks like this:

```
builder.Services.AddDbContext<DataContext>(
    options =>
    {
        options.UseSqlServer(connectionString);
    },
    ServiceLifetime.Scoped
);
```

Console logging

The **options** variable has another method: **LogTo**. This method has a parameter that controls where the log information is sent. You can write to the console output, debug, etc. Let's set it to the console:

```
builder.Services.AddDbContext<DataContext>(
    options =>
    {
        options.UseSqlServer(connectionString);
        options.LogTo(Console.WriteLine);
    },
    ServiceLifetime.Scoped
);
```

The parameter of the **LogTo()** is now an **Action<string>**, which means you can use a method that will be executed when Entity Framework Core wants to log something.

When you run the API, you will see that the console exposes more information when you get all the movies, for example. This is debug information.

```

C:\Users\kenji\source\repos\E  +  ~
0 -> Dictionary<IProperty, int> { [Property: Movie.Id (int) Required PK AfterSave:Throw Val
ueGenerated.OnAdd, 0], [Property: Movie.Description (string) Required, 1], [Property: Movie.GenreId (int?) FK
Index, 2], [Property: Movie.IsDeleted (bool) Required, 3], [Property: Movie.LastModifiedDate (no field, Date
Time?) Shadow BeforeSave:Ignore AfterSave:Ignore ValueGenerated.OnAddOrUpdate, 4], [Property: Movie.Rating (i
nt) Required, 5], [Property: Movie.ReleaseDate (DateTime) Required, 6], [Property: Movie.Seen (bool) Required
, 7], [Property: Movie.Title (string) Required MaxLength(50), 8] }
1 -> Dictionary<IProperty, int> { [Property: Genre.Id (int) Required PK AfterSave:Throw Val
ueGenerated.OnAdd, 9], [Property: Genre.Name (string) Required MaxLength(10), 10] }
SELECT m.Id, m.Plot, m.GenreId, m.IsDeleted, m.LastModifiedDate, m.Rating, m.ReleaseDate, m.See
n, m.Title, g.Id, g.Name
FROM Movies AS m
LEFT JOIN Genres AS g ON m.GenreId == g.Id),
null,
Func<QueryContext, DbDataReader, ResultContext, SingleQueryResultCoordinator, Movie>,
MovieLibrary.Business.DataContext,
False,
False,
True
}',
dbug: 25/06/2024 09:19:02.264 RelationalEventId.ConnectionCreating[20005] (Microsoft.EntityFrameworkCore.Data
base.Connection)
Creating DbConnection.
dbug: 25/06/2024 09:19:02.279 RelationalEventId.ConnectionCreated[20006] (Microsoft.EntityFrameworkCore.Datab
ase.Connection)
Created DbConnection. (15ms).
dbug: 25/06/2024 09:19:02.281 RelationalEventId.ConnectionOpening[20000] (Microsoft.EntityFrameworkCore.Datab
ase.Connection)
Opening connection to database 'MoviesLibrary' on server '(localdb)\MSSQLLocalDB'.
dbug: 25/06/2024 09:19:03.323 RelationalEventId.ConnectionOpened[20001] (Microsoft.EntityFrameworkCore.Datab
ase.Connection)
Opened connection to database 'MoviesLibrary' on server '(localdb)\MSSQLLocalDB'.
dbug: 25/06/2024 09:19:03.325 RelationalEventId.CommandCreating[20103] (Microsoft.EntityFrameworkCore.Databas
e.Command)

```

Debug logging

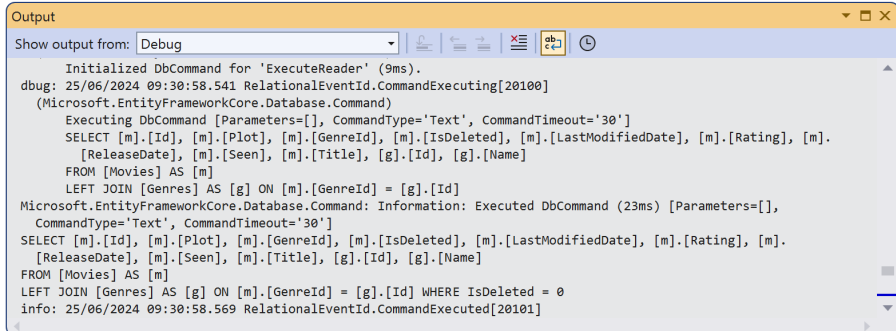
Another way to write and see logs is by using the **Debug** object. When you write to the `Debug.WriteLine()`, the information is written to the output window of Visual Studio. You can redirect the log information from Entity Framework Core to the output window by using `Debug.WriteLine`.

```

builder.Services.AddDbContext<DataContext>(
    options =>
    {
        options.UseSqlServer(connectionString);
        options.LogTo(message => Debug.WriteLine(message));
    },
    ServiceLifetime.Scoped
);

```

The log information is not in the console anymore, but in the output window:



```
Output
Show output from: Debug
Initialized DbCommand for 'ExecuteReader' (9ms).
dbug: 25/06/2024 09:30:58.541 RelationalEventId.CommandExecuting[20100]
(Microsoft.EntityFrameworkCore.Database.Command)
Executing DbCommand [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[IsDeleted], [m].[LastModifiedDate], [m].[Rating], [m].[ReleaseDate], [m].[Seen], [m].[Title], [g].[Id], [g].[Name]
FROM [Movies] AS [m]
LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id]
Microsoft.EntityFrameworkCore.Database.Command: Executed DbCommand (23ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[IsDeleted], [m].[LastModifiedDate], [m].[Rating], [m].[ReleaseDate], [m].[Seen], [m].[Title], [g].[Id], [g].[Name]
FROM [Movies] AS [m]
LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id] WHERE IsDeleted = 0
info: 25/06/2024 09:30:58.569 RelationalEventId.CommandExecuted[20101]
```

This way you keep the output in the console a bit more clean, but the output window shows more than just Entity Framework logging, so it might get a bit messy.

Sensitive data

As you might have noticed with logging, no data is shown. This is because by default Entity Framework Core doesn't show the data because it could contain sensitive data. Exposing data, even in the output window, could cause problems and it readable.

But if you do want to see the data that Entity Framework Core is using, you can enable it by using the **EnableSensitiveDataLogging()** method. This will enable the (sensitive) data. It is part of the **LogTo()** method.

```
builder.Services.AddDbContext<DataContext>(
    options =>
    {
        options.UseSqlServer(connectionString);
        options.LogTo(message => Debug.WriteLine(message))
            .EnableSensitiveDataLogging();
    },
    ServiceLifetime.Scoped
);
```

Again: Be careful with this one. You don't want this to be enabled in a production environment when you work with sensitive data.

Log levels

All types of logging have log levels and they describe the severity of the log-entry. There are a few and the order or severity is important:

- 0 = Trace
- 1 = Debug

- 2 = Information
- 3 = Warning
- 4 = Error
- 5 = Critical
- 6 = None

By default, Entity Framework Core logging is set to Debug (1) and above. That means that debug, information, warning, error, critical, and none are shown. If you want less information, because debug is a bit much, you can set the log level to warning (3).

The log level is a parameter of the **LogTo()** and it's an enum, so you don't have to memorize the numbers.

```
builder.Services.AddDbContext<DataContext>(
    options =>
    {
        options.UseSqlServer(connectionString);
        options.LogTo(message => Debug.WriteLine(message),
LogLevel.Information)
            .EnableSensitiveDataLogging();
    },
    ServiceLifetime.Scoped
);
```

With this setting, you will see all messages with the severity of information, warning, error, critical, and none.

Microsoft logging

Another way of logging with Entity Framework Core is using Microsoft logging, which is the **Microsoft.Extensions.Logging**. For this to work, you need to create a **LoggerFactory** instance. This is a standard factory that can also be used by other parts of your application, but we focus on Entity Framework Core now.

There are a few benefits when using the LoggerFactory; you can reuse the logging mechanism for other parts of your application and you can log to multiple locations at once.

Another big benefit is that the LoggerFactory can be expanded or even overridden. This allows you to write to files or even the event logger of Windows. But I am not going to discuss how to do that since that's not on the Entity Framework Core topic.

Creating LoggerFactory

The LoggerFactory should be created before you can use it in Entity Framework. For example:

```
ILoggerFactory loggerFactory = LoggerFactory.Create(builder => {  
    builder.AddConsole();  
    builder.AddDebug();  
});
```

This instance will create a new LoggerFactory and log to the console and the debug (which is sent to the output window). It does the same as simple logging, but it is a bit more powerful.

To use this logging, simply add it to the configuration with the DbContextOptionsBuilder:

```
builder.Services.AddDbContext<DataContext>(  
    options =>  
    {  
        options.UseSqlServer(connectionString);  
        options.UseLoggerFactory(loggerFactory);  
    },  
    ServiceLifetime.Scoped  
);
```

Sensitive data and log levels

Since we explored these two in the previous chapter I am not going into detail anymore. Instead, I am just showing you how to implement these with the LoggerFactory.

To enable logging sensitive data, enable it on the **DbContextOptionsBuilder**. The log level is set a bit differently. You need to set a minimal log level on the LoggerFactory. Put these two together and the code looks like this:

```
ILoggerFactory loggerFactory = LoggerFactory.Create(builder => {  
    builder.AddConsole();  
    builder.AddDebug();  
    builder.SetMinimumLevel(LogLevel.Debug);  
});  
  
builder.Services.AddDbContext<DataContext>(  
    options =>  
    {  
        options.EnableSensitiveDataLogging();  
        options.UseSqlServer(connectionString);  
        options.UseLoggerFactory(loggerFactory);  
    },  
    ServiceLifetime.Scoped  
);
```

Final words

While there are many ways to log information from your application, including Entity Framework Core, it's best to use the implemented logging functionality of Entity Framework Core.

You can use the simplest way: Using the setting **LogTo()** in the **DbContextOptionsBuilder**. It doesn't need much configuration and you can write the information to a **debug** (output window) or the **console**. With the correct **log level**, you can adjust how much and what information you want to see.

If you want to go a bit further, use the **Microsoft logging**. It needs a bit more configuration, but you get more control. You are also able to (re)use this logging in other areas than Entity Framework Core. You create a logger, which can be used in the application, and connect this logger to Entity Framework Core.

In both ways, you can control if you want to see **(sensitive) data**. But never turn it on on the production server or environment where others can see the information. Also, be careful what you log.

Chapter 11: Inspecting the queries

Data is knowledge and one of the reasons we log certain data. As you have seen at the end of the previous chapter, Entity Framework Core logs information, like it or not.

We can use this information to inspect what Entity Framework Core is doing and how it could affect how we talk to our database.

There are several ways we can inspect the database. We can use inspectors and the `ToQueryString()`.

In this chapter

We want to gather this information to know what our application requests from the database through Entity Framework Core. We use this information to understand which data is used/sent and what kind of SQL queries are used.

But also to optimize the application with queries. If you use databases in the cloud you might notice you not only pay per month but also per transaction. The more transactions you use, the more you pay. It's a good idea to optimize the usage of queries in such a scenario.

Another scenario where you want to inspect the queries is to check what kind of data is queried or inserted into the database. If you get an error the problem might be inside the query rather than in your code.

Query intercepting is one of the most powerful techniques we have when it comes to debugging our applications and learning what those applications are doing.

ToQueryString

The easiest and simplest way to see what kind of query Entity Framework Core is creating is by casting the LINQ statement to a string. Let's rewrite the method `Get(string query)` in the `Movies` class to this:

```

public IEnumerable<Movie> Get(string query)
{
    IQueryable<Movie> result = dataContext.Movies
        .Include(x => x.MovieActors)
        .ThenInclude(x => x.Actor)
        .Include(x => x.Genre)
        .Where(x => x.Title.Contains(query)
            || x.Description.Contains(query));

    return result;
}

```

This way we can have some extra lines of code. To cast a LINQ to a query string, we need to use the method **ToQueryString()** and it works like this:

```
the_linq = linq.ToQueryString();
```

Pretty simple, right? Let's write the query that the method **Get(string query)** creates to the output window.

```

public IEnumerable<Movie> Get(string query)
{
    IQueryable<Movie> result = dataContext.Movies
        .Include(x => x.MovieActors)
        .ThenInclude(x => x.Actor)
        .Include(x => x.Genre)
        .Where(x => x.Title.Contains(query)
            || x.Description.Contains(query));

    Debug.WriteLine("The query is: ");
    Debug.WriteLine(result.ToQueryString());
    Debug.WriteLine("-----");

    return result;
}

```

Running the application (console app or API) will show the query of this method in the output window of Visual Studio.

```
Output
Show output from: Debug
MovieLibrary.ConsoleApp.exe (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared
\Microsoft.NETCore.App\8.0.5\System.Xml.ReaderWriter.dll'. Symbol loading disabled by Include/
Exclude setting.
'MovieLibrary.ConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared
\Microsoft.NETCore.App\8.0.5\System.Private.Xml.dll'. Symbol loading disabled by Include/
Exclude setting.
DECLARE @_query_0_contains nvarchar(50) = N'%shr%';
DECLARE @_query_0_contains_1 nvarchar(4000) = N'%shr%';

SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[Rating], [m].[ReleaseDate], [m].[Seen], [m].
[Title], [g].[Id], [t].[MovieId], [t].[ActorId], [t].[Id], [t].[DateOfBirth], [t].[Name], [g].
[Name]
FROM [Movies] AS [m]
LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id]
LEFT JOIN (
    SELECT [m0].[MovieId], [m0].[ActorId], [a].[Id], [a].[DateOfBirth], [a].[Name]
    FROM [MovieActors] AS [m0]
    INNER JOIN [Actors] AS [a] ON [m0].[ActorId] = [a].[Id]
) AS [t] ON [m].[Id] = [t].[MovieId]
WHERE [m].[Title] LIKE @_query_0_contains ESCAPE N'\ ' OR [m].[Plot] LIKE @_query_0_contains_1
ESCAPE N'\ '
ORDER BY [m].[Id], [g].[Id], [t].[MovieId], [t].[ActorId]

'MovieLibrary.ConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared
\Microsoft.NETCore.App\8.0.5\System.Diagnostics.StackTrace.dll'. Symbol loading disabled by
Include/Exclude setting.
'MovieLibrary.ConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Users\kenji\source\repos
\EntityFrameworkEBook\MovieLibrary.ConsoleApp\bin\Debug\net8.0
\System.Configuration.ConfigurationManager.dll'. Symbol loading disabled by Include/Exclude
setting.
'MovieLibrary.ConsoleApp.exe' (CoreCLR: clrhost): Loaded 'C:\Program Files\dotnet\shared
\Microsoft.NETCore.App\8.0.5\System.Diagnostics.Process.dll'. Symbol loading disabled by
```

It does look a lot more like a stored procedure, but you can see the SELECT and the joins. You can copy this query and paste it into a query window. Don't forget to copy the DECLARE too.

MovieLibrary - SQLQuery1.sql

SQLQuery1.sql

```
1 DECLARE @_query_0_contains nvarchar(50) = N'%shr%';
2 DECLARE @_query_0_contains_1 nvarchar(4000) = N'%shr%';
3
4 SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[Rating], [m].[ReleaseDate], [m].[Seen],
5 [m].[Title], [g].[Id], [t].[MovieId], [t].[ActorId], [t].[Id], [t].[DateOfBirth],
6 [t].[Name], [g].[Name]
7 FROM [Movies] AS [m]
8 LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id]
9 LEFT JOIN (
10     SELECT [m0].[MovieId], [m0].[ActorId], [a].[Id], [a].[DateOfBirth], [a].[Name]
11     FROM [MovieActors] AS [m0]
12     INNER JOIN [Actors] AS [a] ON [m0].[ActorId] = [a].[Id]
13 ) AS [t] ON [m].[Id] = [t].[MovieId]
14 WHERE [m].[Title] LIKE @_query_0_contains ESCAPE N'\ '
15 OR [m].[Plot] LIKE @_query_0_contains_1 ESCAPE N'\ '
16 ORDER BY [m].[Id], [g].[Id], [t].[MovieId], [t].[ActorId]
```

100 % No issues found Ln: 16 Ch: 58 SPC CRLF

T-SQL Results Message

Id	Plot	GenreId	Rating	ReleaseDate	Seen	Title	Id	MovieId	ActorId	Id	DateOfBi
1	A mean lord exiles fairytaile creatures to the swa...	NULL	0	2001-04-22 00:00:00.0000000	0	Shrek	NULL	3	2	2	1963-05-

Query executed successfully at 09:33:08 (localdb)\MSSQLLocalDB (15... | KENJI\kenji (51) | MovieLibrary | 00:00:00 | 1 rows

Now you can investigate what the query is, what it includes (joins), how the searching works, and what the result is.

I wouldn't advise doing this in a production environment. There is no output window, but the code will be executed. Although the user

won't notice it, it does take some performance from the application. You will need to figure out a way to only show the queries when you want.

A downside is that this is hardcoded and the query will be shown each time this method is called. A second downside is that you will have to add the `ToQueryString()` everywhere you are using LINQ on the `DataContext`.

Query interceptors

A query interceptor is a piece of code in our application that can intercept the queries before it is sent to the database. You will only get information about the database(s) you manage in your application/code.

But they are also used to modify or filter queries globally. They allow you to add filtering data based on user roles or add audit information, ensuring consistent behavior across all queries.

Creating a query interceptor isn't hard. All you need to do is override some methods in the `DataContext`, or any class that inherits the `DbContext`. You can also use global query filters.

There are different types of query interceptors, but the one I am looking for is the **command query interceptor**. This one is used to intercept and modify a query or log the query for inspection. A command query interceptor is a class that inherits from **`DbCommandInterceptor`**, which is part of the Entity Framework Core library.

Enough talk, let's build a simple command query interceptor.

Create the class

A command query interceptor is logic and should live inside the application layer, which is the *MovieLibrary.Business*. Let's create a new folder inside that project and call it *QueryInterceptors*. Inside this new folder, create a new class with the name *EFQueryInterceptorLogger*. Make sure the class is public and clean it up.

So far, so good.

Entity Framework Core already has an interceptor, but you don't see

it. If we inherit from that interceptor we can override the needed methods and use them for our own good. That interceptor is called **DbCommandInterceptor** and is part of *Microsoft.EntityFrameworkCore.Diagnostics* namespace.

```
public class EFQueryInterceptorLogger: DbCommandInterceptor
{

}
```

The **DbCommandInterceptor** has a few methods we can override. But the first one we want is **ReaderExecuting()**. It comes in the standard and async versions.

It is called when you want to retrieve information from a database, the **SELECT** queries. This method has a few parameters, which we can use to extract the query.

```
public override InterceptionResult<DbDataReader> ReaderExecuting(
    DbCommand command,
    CommandEventData eventData,
    InterceptionResult<DbDataReader> result)
{

}
```

The parameters are important, so let's break them down:

- **DbCommand**
This one contains all the information about the SQL, the command, and other parameters that are needed to execute the query on the database.
- **CommandEventData**
The **CommandEventData** has information about SQL commands being executed. It includes things like which **DbContext** is running the command, timing information, and specifics of the command itself.
This one is really important for logging and diagnostics. But also for understanding the environment in which database operations occur, tracking problems, and troubleshooting interactions with the database.
- **InterceptionResult<DbDataReader>**
The **InterceptionResult<DbDataReader>** can influence or observe database operations before they are executed. We can intercept commands, maybe modify them, and decide whether to proceed with their execution. The **InterceptionResult<DbDataReader>** is very useful for logging, security checks, or altering queries based on specific

requirements.

If we want to write the *to-be* executed query to our output window, we can write code that looks like this:

```
public override InterceptionResult<DbDataReader> ReaderExecuting(
    DbCommand command,
    CommandEventData eventData,
    InterceptionResult<DbDataReader> result)
{
    Debug.WriteLine($"Executing Query {command.CommandText}");

    return result;
}
```

The reason we need to return the **InterceptionResult** is that it's a chain. If we don't return the correct information (the result of the interception) we break the chain and the process comes to a full stop.

We can change the outcome of the result. But to make sure that change is making it through the end of the chain, we need to return the interception with the change.

Using the interceptor

All that is left is to use the interceptor. We can connect it to the **DataContext** so it will be used for that **DbContext** with the connection string and all. To use the interceptor we need to add it to the options - a type of **DbContextOptionsBuilder** - of the **AddDbContext**, which is located in the *Program.cs* of the API.

Besides adding the type of database and the connection string, the **DbContextOptionsBuilder** has another method: **AddInterceptors**. Yes, multiple. We can add more than one. But let's just stick to one for now.

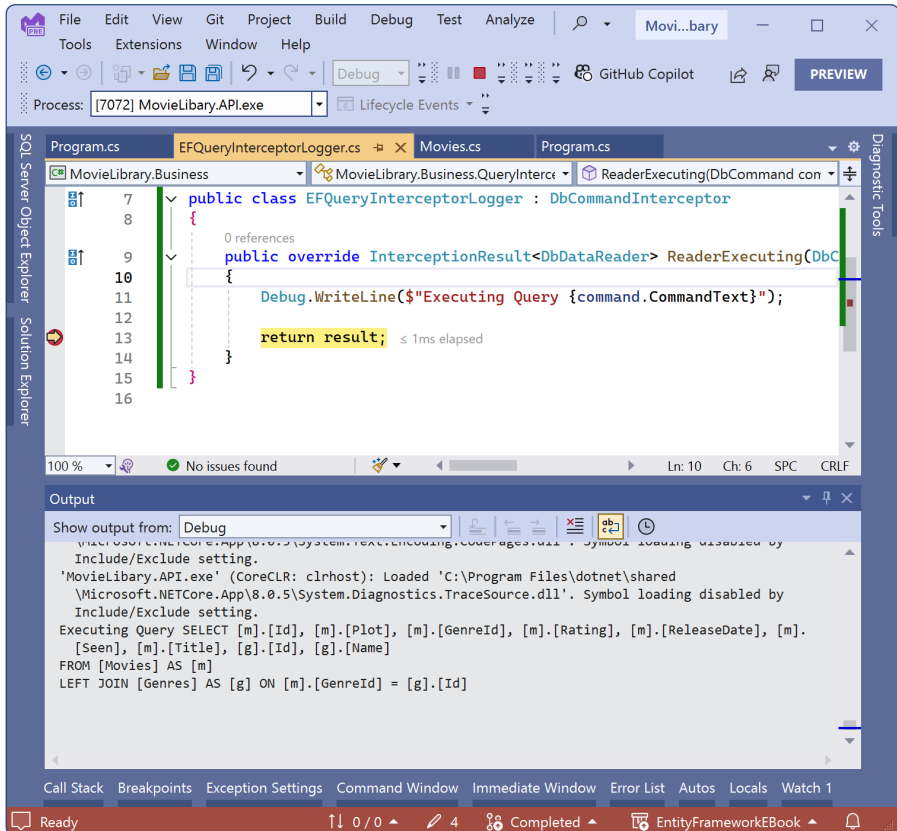
To add and initialize **EFQueryInterceptorLogger** we need to change the code a bit in a way the **DbContextOptionsBuilder** parameter can have more than one setting.

```
string connectionString =
builder.Configuration.GetConnectionString("Default");
builder.Services.AddDbContext<DataContext>(
    options =>
    {
        options.UseSqlServer(connectionString);
        options.AddInterceptors(new EFQueryInterceptorLogger());
    },
    ServiceLifetime.Scoped
```

```
);
```

And that's it!

Let's place a breakpoint in the interceptor. To be more specific on the return result. Run the API (and not the console application, as it did, and then wonder why it's not working...).



Modify the query

One of the cool parts of the interceptor is that you can modify the query if needed.

Preparations

Let's say we want to delete a movie, but not delete-delete it. We want to keep the movie in the database, but it should not be visible at some point. We can add a new property to the Movie entity: **IsDeleted**. This

will be a bool and the default value is false, which is always the case with a boolean.

Go ahead and add the property and update the database accordingly.

Let's use the API to 'delete' a movie. Change the method Delete(int id) in the Movies class so it won't actually delete it, but sets the IsDeleted property to true for the specified movie.

```
public void Delete(int id)
{
    if (id <= 0) throw new ArgumentException($"The {nameof(id)}
    can't be 0 or less.");

    Movie movieToDelete = dataContext.Movies.Single(x => x.Id ==
    id);
    movieToDelete.IsDeleted = true;

    dataContext.SaveChanges();
}
```

Start the API, find the Delete-endpoint, which is the one with the DELETE request, use the id of an existing movie, and execute the request. I am using ID 4, which is the ID for Jaws.

When you get all the movies, the deleted movie is visible too, but we don't want that. To hide the deleted movies we can change the query by using the query interceptor.

Expanding the interceptor

Let's return to the interceptor and change the code in a way that we only get the movies without the deleted movies.

Since an interceptor intercepts all queries that read data we need to make sure only the queries involving movies are changed, not others. We do this by checking which table is used in the SELECT statement.

The **command.CommandText** contains the query, which allows us to check if this contains the part FROM [Movies]. If this is true, we can check if the WHERE is used in the query. If not, we need to add it:

```
public override InterceptionResult<DbDataReader> ReaderExecuting(
    DbCommand command,
    CommandEventData eventData,
    InterceptionResult<DbDataReader> result)
{
```

```
        if (command.CommandText.Contains("FROM [Movies]"))
        {
            if (!command.CommandText.Contains("WHERE"))
                command.CommandText += " WHERE IsDeleted = 0";
            else
                command.CommandText += " AND IsDeleted = 0";
        }

        return result;
    }
}
```

When you run the same API endpoint (the GET) again, you will see all the movies, except the deleted movies.

There is one, big drawback to this: When the original query has an ORDER BY the modified query will be mutilated in a way it can't be used anymore. The WHERE clause will be placed behind the ORDER BY clause, which isn't possible.

You'll have to figure out a way to place everything in the correct order.

Logging the result

Earlier I have shown you how you can use a query interceptor to display the query that will be executed on the database and modify that query if needed. But you can also display the result of the query.

The previous method we have overridden is called **ReaderExecuted()**. It is executed after the database has returned the requested data, or is finished with executing the reader request.

For the next example, I am going to show you how you can grab that returned data, push it in a JSON format, and store it in a file. Since this is a bit of a nice to have, I will go a bit faster through this section than usual.

Create the override

We start by overriding the ReaderExecuted() method in the **EFQueryInterceptorLogger** class.

```
public override DbDataReader ReaderExecuted(
    DbCommand command,
    CommandExecutedEventData eventData,
    DbDataReader result)
```

```
{  
  
}
```

The parameters are basically the same. Again we have a **DbCommand**, which is the same kind as the one we have when overriding the `ReaderExecuting()`.

The **CommandExecutedData** is also somewhat the same as the `CommandEventData`. The **DbDataReader** contains the returned data from the database; the one we want.

We can't just iterate through the `DbDataReader` because if we do the interceptor causes the reader to advance, and when the control returns to the calling code, the `DbDataReader` has no more data to read.

This is easy to fix: Use a `DataTable`.

When the data is copied into a `DataTable`, we can iterate through the rows. We then grab the columns and get the data from the cells per row and per column. We place them all in dictionaries, which represent the columns with values and the rows.

We then combine all the dictionaries and parse it to a JSON file with the filename of the current date and time.

Finally, we return the `DataTable`, since the `DbDataReader` is used and it can't be used again. The `DbDataReader` somewhere inherits the same classes/interfaces as the `DataTable`. Thus you can use a `DataTable` where Entity Framework Core asks for a `DbDataReader`.

The code is rather long...

```
public override DbDataReader ReaderExecuted(  
    DbCommand command,  
    CommandExecutedEventData eventData,  
    DbDataReader result)  
{  
    List<Dictionary<string, object>> rows = [];  
  
    if (result.HasRows)  
    {  
        DataTable dataTable = new();  
        dataTable.Load(result);  
  
        foreach (DataRow dataRow in dataTable.Rows)  
        {  
            Dictionary<string, object> row = [];  
  
            foreach (DataColumn column in dataTable.Columns)
```

```

        {
            row[column.ColumnName] = dataRow[column];
        }
        rows.Add(row);
    }

    JsonSerializerOptions options = new()
    {
        WriteIndented = true
    };

    if (!Directory.Exists("Queries"))
        Directory.CreateDirectory("Queries");

    File.WriteAllText(Path.Combine("Queries",
        $"{DateTime.Now:yyyyMMddHHmmss}.json"),
        JsonSerializer.Serialize(rows, options));

    return base.ReaderExecuted(command, eventData,
        dataTable.CreateDataReader());
    }
    else
    {
        Debug.WriteLine("No rows returned.");
    }

    return base.ReaderExecuted(command, eventData, result);
}

```

Run the application and execute a method that will fire a query. Then check the folder of the API. You can find this folder on your disk by going to the folder where you have cloned the solution with git and then *EntityFrameworkEBook\MovieLibrary.API*.

Inside this folder is a new folder: Queries. Open this one. There should be at least one JSON file. When you open a file you will see it has the result of a query.

If you executed a request that would have fired a SELECT to the database, you will not see the movies that are deleted (property `IsDeleted = false`).

Performance monitoring

Another reason you might want to use query interceptors is to monitor the performance of the queries. A simple stopwatch could help determine how a query is performing.

The idea of this is pretty simple: For each type of query start a stopwatch. When the query is done executing, stop the stopwatch and

mark the result.

There are three kinds of queries you need to monitor: The non-query kind, the reader kind, and the scalar kind. The non-query kind is the query that does not return any rows (create, delete, update). The reader kind does return rows (select, stored procedure, view). The last one, the scalar kind, is for queries that perform calculations or aggregate functions (COUNT, SUM, AVG, MIN, MAX) where the result is a single value.

Each type has its own method to override.

The measurement for the performance should be started before the query is executed. We already used the reader kind:

ReaderExecuting(). The method to override for non-queries is the **NonQueryExecuting()**. The scalar kind uses the method **ScalarExecuting()**.

All three methods also have an async-variant, but I will be using the non-async methods.

But all three methods will be executing a single, private method that will start and stop the stopwatch:

```
private static void LogCommand(DbCommand command)
{
    Stopwatch stopwatch = Stopwatch.StartNew();

    command.Disposed += (sender, args) =>
    {
        stopwatch.Stop();
        Debug.WriteLine($"Command: {command.CommandText}, " +
            $"Execution Time: {stopwatch.ElapsedMilliseconds} ms");
    };
}
```

The **LogCommand** method receives one parameter: **DbCommand**, which is the same type that every to-be-overridden method has. **LogCommand** starts a stopwatch and uses the event **Disposed** to execute a piece of code as soon as the **Disposed** is executed on the **DbCommand**. That piece of code is nothing more than just stopping the stopwatch and writing the information in the output window.

Place this method somewhere at the end of the **EFQueryInterceptorLogger** class.

To use this code, call the **LogCommand** at the very beginning of the **The LogCommand** method receives one parameter: **DbCommand**,

which is the same type that every to-be-overridden method has. LogCommand starts a stopwatch and uses the event Disposed to execute a piece of code as soon as the Disposed is executed on the DbCommand. That piece of code is nothing more than just stopping the stopwatch and writing the information in the output window.

Place this method somewhere at the end of the EFQueryInterceptorLogger class.

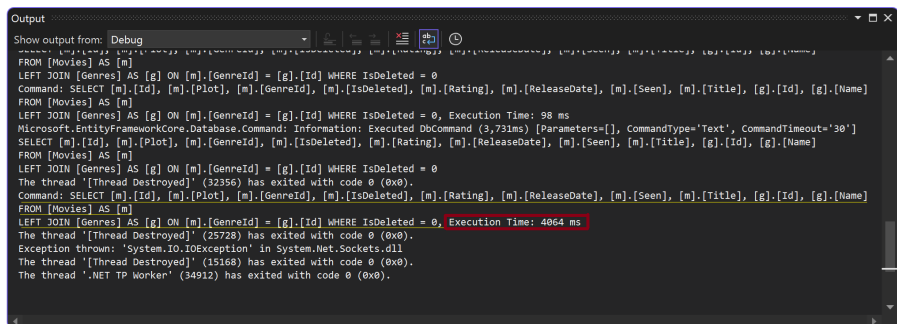
To use this code, call the LogCommand at the very beginning of the ReaderExecuting():

```
public override InterceptionResult<DbDataReader> ReaderExecuting(
    DbCommand command,
    CommandEventData eventData,
    InterceptionResult<DbDataReader> result)
{
    LogCommand(command);

    if (command.CommandText.Contains("FROM [Movies]"))
    {
        if (!command.CommandText.Contains("WHERE"))
            command.CommandText += " WHERE IsDeleted = 0";
        else
            command.CommandText += " AND IsDeleted = 0";
    }

    return result;
}
```

Run the API or the console application and make sure you execute a reader query (SELECT) by getting all the movies. Then check out the output window:



```
Output
Show output from: Debug
FROM [Movies] AS [m]
LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id] WHERE IsDeleted = 0
Command: SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[IsDeleted], [m].[Rating], [m].[ReleaseDate], [m].[Seen], [m].[Title], [g].[Id], [g].[Name]
FROM [Movies] AS [m]
LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id] WHERE IsDeleted = 0, Execution Time: 98 ms
Microsoft.EntityFrameworkCore.Database.Command: Information: Executed DbCommand (3,731ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[IsDeleted], [m].[Rating], [m].[ReleaseDate], [m].[Seen], [m].[Title], [g].[Id], [g].[Name]
FROM [Movies] AS [m]
LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id] WHERE IsDeleted = 0
The thread '[Thread Destroyed]' (32356) has exited with code 0 (0x0).
Command: SELECT [m].[Id], [m].[Plot], [m].[GenreId], [m].[IsDeleted], [m].[Rating], [m].[ReleaseDate], [m].[Seen], [m].[Title], [g].[Id], [g].[Name]
FROM [Movies] AS [m]
LEFT JOIN [Genres] AS [g] ON [m].[GenreId] = [g].[Id] WHERE IsDeleted = 0, Execution Time: 4064 ms
The thread '[Thread Destroyed]' (25728) has exited with code 0 (0x0).
Exception thrown: 'System.IO.IOException' in System.Net.Sockets.dll
The thread '[Thread Destroyed]' (15168) has exited with code 0 (0x0).
The thread '.NET TP Worker' (34912) has exited with code 0 (0x0).
```

Monitor the other overrides

The ReaderExecuting was rather simple since we already have that

one in the interceptor. But this one only monitors the select queries. To monitor the other types we need to override them and call the LogCommand method.

Overriding and monitoring the methods ScalarExecuting() and NonQueryExecuting() are a bit easier since you don't have to change much:

```
public override InterceptionResult<object> ScalarExecuting(
    DbCommand command,
    CommandEventData eventData,
    InterceptionResult<object> result)
{
    LogCommand(command);
    return base.ScalarExecuting(command, eventData, result);
}

public override InterceptionResult<int> NonQueryExecuting(
    DbCommand command,
    CommandEventData eventData,
    InterceptionResult<int> result)
{
    LogCommand(command);
    return base.NonQueryExecuting(command, eventData, result);
}
```

Whatever request you make, each time the database is called via a query and Entity Framework Core, you will get the execution time of the query.

This one saved me once. I had this massive LINQ statement for entities. It was massive because it had several Include-methods and some of the includes had ThenInclude-methods. A lot of entities had relationships with each other and with different conditions.

But then the application was getting really slow. So slow that one request took way too long which made the other requests, who were waiting, get a timeout. I couldn't figure out what it was, so I added performance monitoring to the queries. All queries had a good performance. Except for the big one with the includes.

It turned out that the includes were the problem. To many includes, with ThenIncludes, isn't really good for performance.

I had to split up the LINQ statements. That meant a few more calls extra to the database, but at least my application didn't crash.

Final words on performance monitoring

Although this could come in handy, make sure you run this kind of monitoring only on your own environment or make it really specific in a production environment. If a few 1000 users are using the same query and it keeps monitoring that same query, the performance of the application might go down.

You might want to create some sort of security inside the query interceptors to make sure they work when you want them to.

Security

Security is way more than making sure you keep out unwanted guests. It also involves making sure nothing goes wrong or actions are executed when you don't want them to be executed.

In the previous section, we logged the performance, or execution time, of the queries. I already said we need to be careful not to release this to an environment where a lot of people are using the application (and the queries).

But removing this code when we publish this code and putting it back after publishing isn't a really good idea. Better would be better to add some security in place that will make sure the performance monitors are only running on the development environment.

Environment specific

In case of the performance monitoring, you want to make sure the monitoring only happens in the development environment. When we configured the DbContext in the console application we used the `ASPNETCORE_ENVIRONMENT` to get the current environment. We could use that again:

```
private void LogCommand(DbCommand command)
{
    if
    (Environment.GetEnvironmentVariable("ASPNETCORE_ENVIRONMENT") !=
     "Development")
        return;

    Stopwatch stopwatch = Stopwatch.StartNew();

    command.Disposed += (sender, args) =>
    {
        stopwatch.Stop();
    }
}
```

```

        Debug.WriteLine($"Command: {command.CommandText}, " +
            $"Execution Time: {stopwatch.ElapsedMilliseconds} ms");
    };
}

```

But... This isn't really safe. We have used the same “trick” with the environment variable in the console application, but we called it `ENVIRONMENT`, not `ASPNETCORE_ENVIRONMENT`. If you want to use the environment variables for this, be sure to use the same name.

It would be better to create a configuration setting for this and use the option pattern. We already have the `appsettings.json`. Let's add a new section with a value:

```

{
  ...
  "InterceptorSettings": {
    "Performance": false
  }
}

```

Next, create a new class inside the `MovieLibrary.Domain`, folder `Models`. Call the class, and the file, **InterceptorSettings**. Add the following code:

```

public class InterceptorSettings
{
    public bool Performance { get; set; }
}

```

Go to the `Program.cs` of the API. Use the option pattern to get the section “`InterceptorSettings`” from the app settings and parse it to the **InterceptorSettings** class. Then add that variable to the location where we initialize the interceptor:

```

...
string connectionString =
builder.Configuration.GetConnectionString("Default");
InterceptorSettings interceptorSettings = builder.Configuration
    .GetSection("InterceptorSettings")
    .Get<InterceptorSettings>();

builder.Services.AddDbContext<DataContext>(
    options =>
    {
        options.UseSqlServer(connectionString);
        options.AddInterceptors(new
EFQueryInterceptorLogger(interceptorSettings));
    },
    ServiceLifetime.Scoped
);

```

All that remains is creating a primary constructor (or regular

constructor if you want) on the **EFQueryInterceptorLogger** class and add the **InterceptorSettings** class and use it in the **LogCommand**:

```
public class EFQueryInterceptorLogger(InterceptorSettings settings)
: DbCommandInterceptor
{
    ...

    private void LogCommand(DbCommand command)
    {
        if (!settings.Performance)
            return;

        Stopwatch stopwatch = Stopwatch.StartNew();

        command.Disposed += (sender, args) =>
        {
            stopwatch.Stop();
            Debug.WriteLine($"Command: {command.CommandText}, " +
                $"Execution Time: {stopwatch.ElapsedMilliseconds}
ms");
        };
    }
}
```

Final words

To quickly see what kind of query is sent to the database, you can use something as simple as the **ToQueryString()** method. This will show the query that will be executed on the database. But it comes with some downsides: hardcoded and always executed.

Inspecting the queries helps to get insights into the queries your application, or Entity Framework Core, creates and fires towards the database. But also to modify when you have specific wishes, like showing a subset of the data by adding a WHERE clause to the query.

By creating a class and inheriting from **DbCommandInterceptor** you can create query interceptors to **monitor**, **log**, and **modify queries** before they are sent to the database. But you can also inspect the result when the query is executed and the database has sent information back.

The **DbContextOptionsBuilder** has a method called **AddInterceptors**, which allows us to simply add an interceptor and use it. Inside the interceptor, you can determine if the interceptor should be used or not. You can do this by checking the environment of using a setting in the app settings.

But be careful where and how you use them. Extensive use of such techniques could decrease the performance of your application.

Chapter 12: Tracking

We already saw the method `SaveChanges()`; it commits the changes you made from your data to the database. You need to use `SaveChanges` when you want to create, delete, or update data from your application to the database.

`SaveChanges()` works with tracking. This is like a system that keeps an eye on the changes you make to data objects in your application.

When you retrieve data from the database, Entity Framework Core tracks the objects and their state (added, modified, or deleted). If you change an object, Entity Framework Core notices the change. When you save changes, Entity Framework Core knows what to update in the database.

In this chapter

Entity Framework Core **tracking** helps ensure your data stays consistent without you having to manually track changes.

When you grab information from the database through entities, Entity Framework Core tracks those entities, making it really easy to submit changes back to the database.

Although Entity Framework Core tracking is really powerful, it does decrease the performance a bit. You won't really notice it, but the more people use the application, the more the decreased performance is noticeable. To help you, you can **turn off the tracking** of entities when needed.

When you retrieve information from outside your application, like input from a user, you can put that information into an entity and **attach** it to Entity Framework Core, making it track the new entity.

Stored procedures are not entities per se, as we have learned in a previous chapter. But we can keep track of the information retrieved from a stored procedure. Thus, the subject stored procedure will come back in this chapter.

Tracking entities

If we look at the following code:

```
public Movie Get(int id) => dataContext.Movies.Single(x => x.Id == id);
```

This is pretty simple: The `Get()` method returns a single movie that has the same id as the value of the parameter `id`. If no movie is found, an exception is thrown. But we are going to assume a movie is found.

That movie, that is returned, is being tracked by Entity Framework Core. As soon as we change a value of one of the properties and call the `SaveChanges()` on the `DataContext`, that change (or multiple) is saved to the database.

An example of this can be seen in the `Update()` method of the `Movies` class. The method gets a single movie and then changes some properties. When the values are set, the `SaveChanges()` is called and the changes are sent to the record in the table of the database.

To demonstrate how powerful this is, we are going to change the mapping `GetMovieById()` of the API.

Store the single movie retrieval in a variable. Add 1 to the rating of the movie before returning it.

```
app.MapGet("/api/movies/{id:int}", (int id, IMovies movieService)
=>
{
    Movie movie = movieService.Get(id);
    movie.Rating++;

    return movie;
})
.WithName("GetMovieById")
.WithOpenApi();
```

Run the API and retrieve a single movie through the correct endpoint a few times. Nothing happens, right?

Let's inject the **DataContext** in the mapping and save the changes after we have added 1 to the rating:

```
app.MapGet("/api/movies/{id:int}", (int id, IMovies movieService,
DataContext dataContext) =>
{
    Movie movie = movieService.Get(id);
    movie.Rating++;
    dataContext.SaveChanges();

    return movie;
})
```



```
.WithName("GetMovieById")  
.WithOpenApi();
```

Run the API again and retrieve a single movie through the correct endpoint a few times. You can now see that the rating is always upped with one each time you request a single movie.

Even though the single movie is retrieved through Entity Framework Core from a different class, the movie is still being tracked.

If you typed along, **undo the changes**, because it doesn't make sense to up the rating each time we get a movie.

Attaching entities

The previous section gave some idea about how Entity Framework Core tracks the entities or rather the data of the entities. It's a basic example, but we should seriously take a look at the current update code of a movie. Because it's not really correct.

We already have a full movie, with all the required fields. So... Why do we get a movie from the database and then update those fields? Can't we connect the parameter movie to an entity?

This is the current update method inside the Movies class.

```
public void Update(Movie movie)  
{  
    ArgumentNullException.ThrowIfNull(movie);  
  
    if (movie.Id <= 0) throw new ArgumentException($"The  
{nameof(movie.Id)} can't be 0 or less.");  
  
    Movie movieToUpdate = dataContext.Movies.Single(x => x.Id ==  
movie.Id);  
  
    movieToUpdate.Seen = movie.Seen;  
    movieToUpdate.Title = movie.Title;  
    movieToUpdate.Description = movie.Description;  
  
    dataContext.SaveChanges();  
}
```

The answer to the question if we can connect a certain type to an entity from the database is yes... I wouldn't have started about it if it wasn't possible.

Anyway... Entity Framework Core has something called **Attach** and

you can use it on an entity. The only condition to do this is that the key field has a value (the id) and all the required fields have values.

Then we can really clean up the Update method:

```
public void Update(Movie movie)
{
    ArgumentNullException.ThrowIfNull(movie);

    if (movie.Id <= 0) throw new ArgumentException($"The
{nameof(movie.Id)} can't be 0 or less.");

    dbContext.Movies.Attach(movie);
    dbContext.Entry(movie).State = EntityState.Modified;
    dbContext.SaveChanges();
}
```

To take the **movie** parameter and attach it to the Movies, we attach it to Entity Framework Core. From this point on, Entity Framework Core is tracking this object, which has become an entity.

But we also need to tell Entity Framework Core what the current state of this entity is. We do this by setting the **EntityState**.

There are a few states: Modified, detached, added, unchanged, and deleted. Each with its own result. Most of the states are self-explanatory.

But what happens if we use an ID that doesn't exist? If you try to update a movie with the ID that does not exist in your database, 99 in my case, you will get an error:

The database operation was expected to affect 1 row(s), but actually affected 0 row(s); data may have been modified or deleted since entities were loaded.

This error is logical because it tries to attach an entity that doesn't exist.

But what if we keep using that id, but change the entity state to **Added**?

An error occurred while saving the entity changes. See the inner exception for details.

---> Microsoft.Data.SqlClient.SqlException (0x80131904): Cannot insert explicit value for identity column in table 'Movies' when IDENTITY INSERT is set to OFF.

Also logical, because an ID, or primary key, can only be set by the database, but through code.

But... What happens if we remove the check on the id (≤ 0) and remove the id from the update request?

That works! It will create a new entity and store the data inside the database.

We can conclude from this: We can create a **CreateOrUpdate()** method!

Let's create a new method inside the **Movies** class (don't forget the interface!). Call the new method **CreateOrUpdate()**.

Inside this method, we want to check if the id of the movie is set. If so, attach the movie and set its state to modified. If the id is 0, attach the movie, but set the state to "added". Pretty simple, right?

```
public void CreateOrUpdate(Movie movie)
{
    dataContext.Movies.Attach(movie);

    dataContext.Entry(movie).State = movie.Id == 0
        ? EntityState.Added
        : EntityState.Modified;

    dataContext.SaveChanges();
}
```

In the **Program.cs** of the API, we can change the mappings for creating and updating:

```
app.MapPost("/api/movies/", (Movie movie, IMovies movieService) =>
{
    movieService.CreateOrUpdate(movie);
})
.WithName("CreateMovie")
.WithOpenApi();

app.MapPut("/api/movies/", (Movie movie, IMovies movieService) =>
{
    movieService.CreateOrUpdate(movie);
})
.WithName("UpdateMovie")
.WithOpenApi();
```

When this is done, we can remove the **Update()** and **Create()** methods from the **Movies** class, and the interface **IMovies**.

We have fully implemented the tracking state of entities for updating

and creating.

No tracking

Tracking entities is great if you want to easily change the data. But in some cases, you don't have to track an entity. If you are just getting the information and exposing it to the front end of the application, tracking could be costly performance-wise.

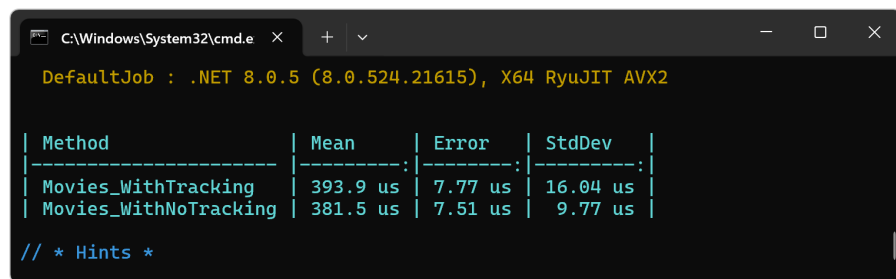
The Movies class has 3 get methods: The get all movies, the get by id, and the get for searching. All 3 are only retrieving information and exposing it to the world. None of them is being used for something important for tracking.

Let's take the **Get(int id)**. We can disable the tracking, disconnecting the entity from Entity Framework Core. The only thing we have to do is use **AsNoTracking()** on the Movies:

```
public Movie Get(int id) =>
    dbContext.Movies.AsNoTracking().Single(x => x.Id == id);
```

If you want to change the data in the found entity, you can, but when you call **SaveChanges()** on the **DataContext**, nothing will be sent to the database, because the entity is not attached.

Using the **AsNoTracking()** detaches the entities from Entity Framework Core, making them not able to change the data in the database. This does help the performance of your application a little bit, as you can see in the next image, which is a screenshot from a benchmark I did with the movies:



Method	Mean	Error	StdDev
Movies_WithTracking	393.9 us	7.77 us	16.04 us
Movies_WithNoTracking	381.5 us	7.51 us	9.77 us

// * Hints *

It's not much, but the standard deviation (stdDev) is lower with no tracking, which is better.

By default, all the entities you retrieve through Entity Framework Core are tracked, but you can change this behavior. There is a setting

in the options builder which you can use to turn off tracking by default. It's called the **UseQueryTrackingBehavior**.

The **UseQueryTrackingBehavior** is set to tracking by default and if you want to change that to no tracking at all, you want to change this in the options builder like this:

```
builder.Services.AddDbContext<DataContext>(
    options =>
    {
        options.UseSqlServer(connectionString);
        options.AddInterceptors(new
            EFQueryInterceptorLogger(interceptorSettings));
        options.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
    },
    ServiceLifetime.Scoped
);
```

Each entity you now retrieve is not tracked by Entity Framework Core. If you want to send data back to the database, you will need to re-attach the entity before you do so.

Identity resolution

Entity Framework Core uses something called identity resolution. It means that Entity Framework Core will make sure that each entity (object) is unique and consistent.

If you don't use identity resolution and you query the same movie twice, you might get two different objects representing that movie. Changing one and saving it will cause the other one to still have old data.

But with identity resolution: If you query the same movie twice, you get the same object. If you change it and save it, all references to that movie will have that new data.

If you use no tracking that means you don't use identity resolution and you could run into problems when you have multiple objects of the same data, which are in essential entities.

If you still want to disable the tracking but keep the identity resolution, you can use the

QueryTrackingBehavior.NoTrackingWithIdentityResolution. You still disable the tracking, but keep the identity resolution.

```
builder.Services.AddDbContext<DataContext>(
    options =>
```

```

    {
        options.UseSqlServer(connectionString);
        options.AddInterceptors(new
            EFQueryInterceptorLogger(interceptorSettings));
        options.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
    },
    ServiceLifetime.Scoped
);

```

Tracking stored procedures

In a previous chapter, we talked about how we can retrieve data from executing stored procedures. One stored procedure isn't using an entity and we created a special model for it and connected that model with the stored procedure, without using a primary key.

The other stored procedure does use an entity and that entity is the **Movie** entity. We did not connect this stored procedure with the entity, because the connection is already there.

Awesome, why am I telling you this? Because we are learning about tracking and although it doesn't look like it could work, stored procedures can track the entities too.

The first stored procedure, the `GetMovies()`, isn't an entity in a way it is related to any data with a primary key in the database. We can track it, but it doesn't store any data in the table.

The second stored procedure, the `Search()`, uses the entity **Movie** as it is explicitly connected to it:

```

public IEnumerable<Movie> Get(string query)
{
    SqlParameter queryParam = new SqlParameter("@query", query);
    return dbContext.Movies.FromSqlRaw("EXEC Search @query",
        queryParam).ToList();
}

```

That means that as soon as we retrieve the information from the stored procedure, we can change an entity (a single movie) and save the changes. Here is a small example:

```

public IEnumerable<Movie> Get(string query)
{
    SqlParameter queryParam = new SqlParameter("@query", query);
    List<Movie> movies = dbContext.Movies
        .FromSqlRaw("EXEC Search @query", queryParam).ToList();

    movies[0].Rating = 99;
    dbContext.SaveChanges();
}

```

```
    return movies;  
}
```

Run the API and search for a movie. You will notice that the first movie will get a new rating of 99.

If you are typing along, please **undo these changes**. They are for demonstration purposes only and don't have any logical functionality.

Final words

Tracking entities helps us keep track of any changes we make to entities. We don't have to write whole update queries to store a single change; Entity Framework Core does that for us.

If we get an entity from an external source, like the input of a user or the API, we can use **Attach < >** to attach the data to an entity, making Entity Framework Core track it for us. Ideal for creating or updating, because Entity Framework Core can figure out if the entity needs to be created or updated.

Although the tracking does come with a minor performance decrease, we can turn it off by using **NoTracking()** when retrieving the data from the database.

The downside of **NoTracking()** is that we lose the identity resolution. This is to keep the objects unique. But we can use the **QueryTrackingBehavior.NoTrackingWithIdentityResolution** to use the **NoTracking()**, but keep the identity resolution.

And finally, we talked about **stored procedures**... Again. Although the information from a stored procedure isn't tied to an entity per se, we can still attach it and store information. This is only when the data from the stored procedure can be mapped to an entity.

Chapter 13: Fluent API

We already talked about it many times in this book, but let's talk about Fluent API for a short while.

Fluent API in Entity Framework Core is a way to configure your database and entities using method chaining instead of attributes. It lets you specify relationships and mappings between your entities and the database tables with C# code.

In some cases, it's easier to use the attributes and sometimes it's better to use Fluent API. While most subjects we discussed are doable by the way we did, they can also be done by using Fluent API.

Still, there are a few things you can only do with Fluent API.

In this chapter

Fluent API is a pretty standard functionality with Entity Framework Core and it's there to help you set up your entities and database model. In some cases, it can be used instead of the attributes we have used earlier in the chapter *Setting up entities*.

The Fluent API is a way to build and configure entities in a readable way. It uses method chaining to create a flow that almost reads like natural language, making it easy to understand and maintain. Instead of setting properties one by one or using complex configurations, you call a series of methods in a specific order, each returning the object itself. This allows you to stack these calls together smoothly.

In this chapter, we are going to look at the attributes **Required**, **MaxLength**, and **NotMapped** and (re)write them with the **Fluent API**. We will also take a look at how to set **unique indexes** with the Fluent API.

However, some subjects can only be done with Fluent API, such as **shadow properties**; properties in the database table, but not in the entity.

The Fluent API is used inside the **OnModelCreating** method, so you do need to have that method overridden in the **DataContext** class. We will use the **modelBuilder** to configure and set up the Fluent API on

the entities and database.

We already used the Fluent API a lot and we used it to create the **relationship** between a Movie and MovieActors. And create a **composite key** for the entity MovieActor between the properties MovieId and ActorId.

Seeding (**HasData()**) is also part of the Fluent API.

This chapter is to dive a little bit deeper into the Fluent API and show you that little bit extra. It is to give you an idea of what it is and how it works. Do know that you can do much more with it, but I will just show you the basics.

Setting up entities

Yes, we already covered this subject, but with attributes. Let's look at it again, but now with Fluent API.

The idea behind Fluent API is to configure your entities for your database with C# code and chaining.

Required

If you want to set a property of an entity to required and with a max length, you use the following code:

```
[MaxLength(50)]  
[Required]  
public string Title { get; set; }
```

But you can also use Fluent API for this if you prefer. It would look like this:

```
modelBuilder.Entity<Movie>().Property(x => x.Title)  
    .IsRequired()  
    .HasMaxLength(50);
```

This does the same thing but in C# code and chaining.

What it does is use the **modelBuilder** and then select the entity we want to set up, Movie in this case. Then we select the property with the **Property** method. By using a lambda expression we select the property of the Movie entity. We then have a lot of extensions we can use to configure that property.

There are way too many options to discuss here so I suggest you take a look at it. Also, this chapter is to give you an idea about Fluent API, on how it works so you can recognize it in the future.

Just as with attributes, you need to add a new migration and update the database for the changes to take effect.

NotMapped

The NotMapped attribute allows you to use properties in an entity without having that property created as a field in the database table. While most Fluent API methods are named pretty much the same as with the attributes, the NotMapped attribute is a bit different. It's not called NotMapped(), and it's also not an extension of Property. It's an extension of the entity itself and it's called **Ignore()**.

So this:

```
[NotMapped]
public string? UrlFriendly { get; set; }
```

Is this with Fluent API:

```
modelBuilder.Entity<Movie>().Ignore(x => x.UrlFriendly);
```

As you can see, not all Fluent API functions are set on a property, but sometimes on an entity.

I will keep the attribute in the completed type-along project.

Indexes

An index in a database is a data structure that improves the speed of data retrieval operations on a table at the cost of additional space and write operations. It is created to enhance the performance of queries by providing a more efficient way to look up and access data.

In the earlier version of Entity Framework (before the Core variant) we could set an attribute on the property of the entity to make it an index. But since Entity Framework Core, this isn't possible anymore and it has to be done by using Fluent API.

Indexes still increase performance and should still be used. But to create them you need to set them with Fluent API.

Let's put an index on the Title of the entity Movie. We can't select the

entity and then the property Title. We set an index on the entity:

```
modelBuilder.Entity<Movie>().HasIndex(x => x.Title);
```

Add a migration and update the database. This will create the index on the Title. You can't continue on this line to set the required and max length, so you need to set those on a separate code line.

The index increases performance and much more. But you can also make an index unique, meaning that the value of the index field should be unique in the table. A bit like the primary key.

```
modelBuilder.Entity<Movie>()  
    .HasIndex(x => x.Title)  
    .IsUnique();
```

In the past, before Entity Framework, you had to rebuild the indexes on a database. With Entity Framework Core you don't have to. Entity Framework Core will take care of this for you.

Shadow properties

Some fields of a table don't have to be in an entity. Audit information for example should be in the table of the database but is not necessarily needed in the C# entity to work with. This is a bit the opposite of the **[NotMapped]**.

Let's add some simple audit information; each time a movie is edited we want to set the date and time when that movie is edited. Great for when you have a specific problem with a movie and you can see when it was the last time it has been changed.

Creating the property

We can't create the property inside the entity Movies; this beats the whole idea of what I just wrote.

Instead, we need to create a property that is there, but is not really there: It's a shadow. We add this through the fluent API, placing the property on the entity model.

Inside the **OnModelCreating()**, located in the DataContext, we point to an entity. We then use the **Property()** method and assign a data type. The Property() method has a parameter that is the name of the field inside the table.

```
modelBuilder.Entity<Movie>().Property<DateTime?>("LastModifiedDate");
```

When you create a new migration, the LastModifiedDate will be added to the table Movies inside your database.

Starting the API and retrieving a movie or all movies will show all the properties in the entity, but not the shadow properties.

Setting a value

A shadow property isn't in your entity, we already know that. When you retrieve a single movie you don't have access to the property/field LastModifiedDate. Accessing shadow properties works a bit differently.

An entity doesn't have the shadow property, we already know that. When you retrieve a single movie you don't have access to the property/field LastModifiedDate. Accessing shadow properties works a bit differently.

To set the value for LastModifiedDate we need to grab it through the Property() method and use the **CurrentValue** on that property. Let's update the value of LastModifiedDate when a movie is updated.

```
public void CreateOrUpdate(Movie movie)
{
    dbContext.Movies.Attach(movie);

    dbContext.Entry(movie).Property("LastModifiedDate").CurrentValue
= DateTime.Now;

    dbContext.Entry(movie).State = movie.Id == 0
        ? EntityState.Added
        : EntityState.Modified;

    dbContext.SaveChanges();
}
```

You can run the API now. Find a movie to update and send the PUT request. Try to retrieve that movie again in the API:



The LastModifiedDate isn't showing in the result. Let's view the data of the table through the SQL Server Object Explorer.

Id	Title	ReleaseDate	Seen	Rating	Plot	GenreId	IsDeleted	LastModifiedDate
1	The Hunger Ga...	21/03/2012 00...	False	7	Katniss volunte...	NULL	False	21/06/2024 09...
2	The Matrix	31/03/1999 00...	True	8	When a beautif...	NULL	False	NULL
3	Shrek	22/04/2001 00...	False	0	A mean lord exi...	NULL	False	NULL
4	Jaws	20/06/1975 00...	True	7	When a killer s...	NULL	True	NULL
5	The Muppets ta...	13/07/1984 00...	True	10	Kermit and his f...	NULL	False	NULL
6	The Hunger Ga...	21/03/2012 00...	False	2	Katniss volunte...	NULL	False	NULL
* NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

And there it is!

But updating the property by code isn't convenient. There are some ways to set default values, but they only work when you create a new entity. To update the value when you update an entity, you should override the SaveChanges(), but that is a completely different story.

To read the value of the field in C#, if needed, you can also use the **CurrentValue**.

Final words

The Fluent API is a great way to programmatically set up entities and the database. We write the code in a readable way which makes it easier to understand for others. You can do almost everything with Fluent API you can also do with attributes. However, some subjects can only be done through the Fluent API, like relationships, composite keys, and more.

In this chapter, I have shown you how you can use **Fluent API** instead of the attributes shown in an earlier chapter. But also how you can add **indexes** to your data tables.

Shadow properties are perfect for storing information that shouldn't

be shown in the application or exposed to the outside. This way, shadow properties allow us to keep the entity classes clean while still maintaining additional information in the database.

You can use Fluent API for many more things, but this book is to give you a headstart and some idea on how the basics of Entity Framework Core work. This chapter should give you enough information to find your way on the Internet.

Chapter 14: Unit testing

Unit testing is, in my opinion, one of the most important tasks of a developer. With testing, we make sure our code is stable and works. Not only when we created the code, but also when we change the code that might affect the outcome.

When you are working with Entity Framework Core and injecting the `DataContext`, like we have so far, testing becomes a bit annoying. The `DataContext` is injected and we don't have a while service builder when we test. I mean we could... But we shouldn't.

Another thing with unit testing is that we don't want to rely on 3rd parties, like a file system, API(s), or databases. How can we create unit tests and still have Entity Framework Core implemented and not use the data from the database?

In this chapter

When we work with unit tests we want to test our code and our code only. We don't want to keep in mind there could be an influence from the outside, like data in a database or data from an API.

In this chapter, we are going to dive a little bit into the world of unit testing and Entity Framework Core. How can we test methods in a class that depends on Entity Framework Core? And even better: How can we have data from Entity Framework Core without actually calling a database?

The answer is simple; **in-memory databases**. The API and the console application are both configured to use SQL Server, but in-memory is a provider too. This means we can change the provider from SQL Server to an in-memory database without the need to change entities or other stuff involving Entity Framework Core.

But an in-memory database does come with some flaws: There is **no auto-increment**. That means we need to create something that Entity Framework Core will create the IDs when we use the in-memory database and let the database create the IDs when the provider is something else.

I will not explain how to create all the tests, but just enough so you

can continue on your own.

This chapter is not only to show you how to unit test with Entity Framework Core implemented, but also how you can use the in-memory database with Entity Framework Core.

Setting up the test project

To create and run unit tests I will be using **xUnit**. NUnit will work too, but I find (personal opinion) xUnit a bit nicer to work with. Let's create a xUnit project. Add it to your solution and name it `MovieLibrary.Business.Tests`. The reason for this name is that you want to unit test the application code, not the front end. The domain is also not logical to test. If you have more application projects, you can separate them this way. Make sure you use .NET 8 since all the other projects are .NET 8 too.

Next, make sure to make a project reference to the *MovieLibrary.Business* and the *MovieLibrary.Domain*.

There is already a test class present. You can rename that one or remove it and add a new class. Whatever you choose, call it **MoviesTests**, since this class will be all about testing movies.

And that's it for this section!

In-memory

The biggest problem we face is that Entity Framework Core needs to have a connection to a database, but we don't want that. We don't want to connect it to a database that changes all the time during development... Or worse; during usage. We need something that Entity Framework Core will be using during testing. Something that doesn't change all the time. And that's in-memory databases.

At the very beginning of this book, I told you that Entity Framework Core can communicate with different database types (Oracle, MySQL, MSSQL, etc.). You don't have to change any code, only the configuration of the DbContext which server type to use in the service container. This is, in our case, **UseSqlServer**.

But there is another database type: In-memory. Which is exactly what it is: A database in memory. This gives us complete control over the

data and is perfect for (unit) testing. Another big advantage is that the data in the in-memory database is reset as soon as you restart the unit tests.

To make use of the in-memory database we need to install a package: *Microsoft.EntityFrameworkCore.InMemory*. Install this package in the xUnit project now.

Setting up the database

Now we can set up an in-memory database inside the test class and all starts with a constructor. Inside this constructor, we initialize the `DataContext`, but this one has a parameter: `DbContextOptionsBuilder`. We need to create this one first and the `DbContextOptionsBuilder` contains the setting of where to find the database, which will be the in-memory database.

```
public MoviesTests()
{
    DbContextOptions<DataContext> options = new
    DbContextOptionsBuilder<DataContext>()
        .UseInMemoryDatabase(databaseName: "InMemoryDatabase")
        .Options;
    DataContext dataContext = new DataContext(options);
}
```

We now have an initialized `DataContext`, read to consume some data.

The **UseInMemoryDatabase** is the equivalent of **UseSqlServer** but doesn't get a connection string. Instead, it has a parameter **databaseName**, which is literally the name it will be using. You can put in here whatever you want. Just keep it simple and easy to remember.

Now we can add data to the in-memory database. This works the same as the seeding and I highly suggest you copy-paste it from the `DataContext.OnModelCreating`. But instead of `modelBuilder.Entity<Movie>().HasData()` use **`dataContext.Movies.AddRange()`**:

```
public MoviesTests()
{
    DbContextOptions<DataContext> options = new
    DbContextOptionsBuilder<DataContext>()
        .UseInMemoryDatabase(databaseName: "InMemoryDatabase")
        .Options;
    DataContext dataContext = new DataContext(options);

    dataContext.Movies.AddRange(
```

```

        new Movie()
        {
            Id = 1,
            Description = "Katniss volunteers to replace her sister
in a tournament that " +
                "ends only when one participant remains. Pitted
against contestants " +
                "who have trained for this all their life, she has
little to rely on.",
            ReleaseDate = new DateTime(2012, 03, 21),
            Seen = false,
            Title = "The Hunger Games"
        },
        new Movie()
        {
            Id = 2,
            Description = "When a beautiful stranger leads computer
hacker Neo to a forbidding " +
                "underworld, he discovers the shocking
truth--the life he knows is the elaborate" +
                " deception of an evil cyber-
intelligence.",
            ReleaseDate = new DateTime(1999, 03, 31),
            Seen = true,
            Rating = 8,
            Title = "The Matrix"
        }
    );
    dataContext.SaveChanges();
}

```

Don't forget the `dataContext.SaveChanges()`! Even though this is an in-memory database, you still need to submit the changes.

A first test

Now that we have the database configuration, we can create a simple test. Let's test the `Movies.Get()` and check if it returns all the movies currently in the database.

Initialize the `Movies` class in the constructor and store it in a private read-only field in the class:

```

public class MoviesTests
{
    private readonly Movies moviesService;

    public MoviesTests()
    {
        ...
        moviesService = new Movies(dataContext);
    }
}

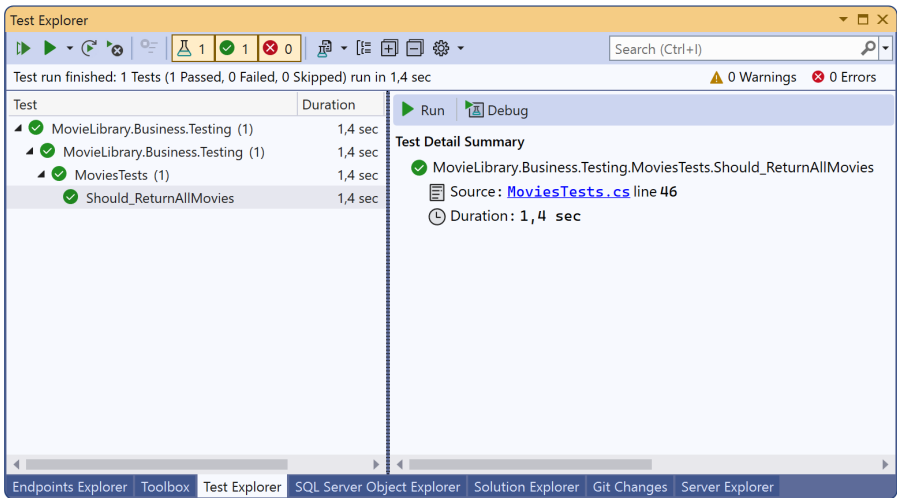
```

Next, create a test method with the name **Should_ReturnAllMovies()**, and call the **movieService.Get()** and assert that the number of movies equals 2 (or another amount, if you have more or fewer movies).

```
[Fact]
public void Should_ReturnAllMovies ()
{
    List<Movie> movies = moviesService.Get().ToList();

    Assert.Equal(2, movies.Count);
}
```

Run the test in the Test Explorer and behold the result:



Adding a movie

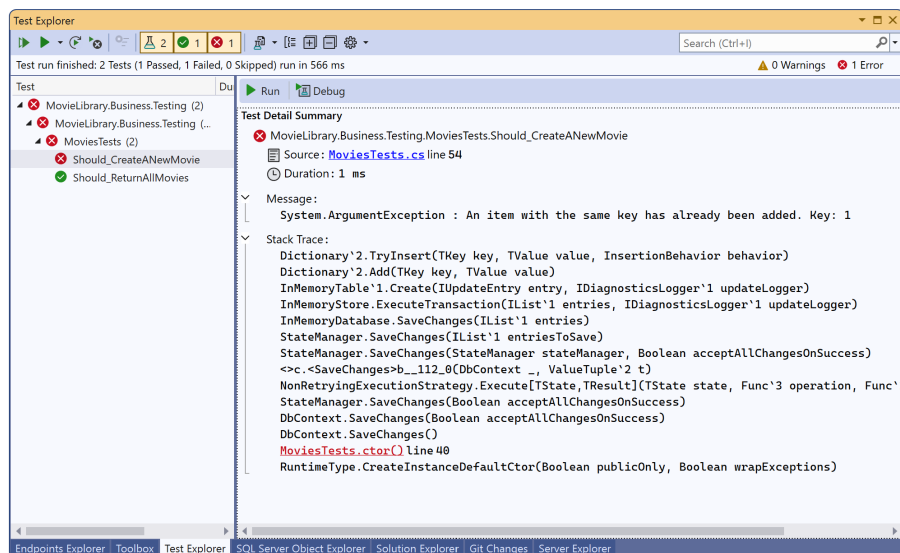
Let's test the creation of a movie through the **Movies** class. Since we already know how this works and this is not a xUnit book... Here's the code:

```
[Fact]
public void Should_CreateANewMovie ()
{
    moviesService.CreateOrUpdate(new Movie()
    {
        Description = "A mean lord exiles fairytale creatures to the swamp of a grumpy ogre, who must go " +
                    "on a quest and rescue a princess for the lord in order to get his land back.",
        ReleaseDate = new DateTime(2001, 04, 22),
        Seen = false,
        Title = "Shrek"
    });
}
```

```
Assert.Equal(3, moviesService.Get().Count());
```

```
}
```

But when you run this test you get a big error, saying there is already an item with the same key 1.



When using an in-memory database with Entity Framework Core, the primary key auto-increment does not have the same behavior as when using a SQL Server database. The in-memory database provider does not support auto-increment for primary keys by default.

We can do two things: Add the ID ourselves or write a piece of code that will help us.

I want to go with the second option to show you how you could do this.

Creating the Id for in-memory only

This is a great moment to override the `SaveChanges()`, or the `SaveChangesAsync()` if you want. Because we need to take control of the auto-increment of the ID, we need to calculate it when the `SaveChanges` is executed.

Inside the override, we need to check the provider, which is either SQL or in-memory. If it's not in memory, we check all the Movies in the **ChangeTracker** if there are movies to add. If so, we set the id and continue saving the changes.

To keep track of the IDs, we create a private read-only field at the top of the DataContext class.

```
private static int _movieIdCounter = 1;
...
public override int SaveChanges()
{
    if (Database.ProviderName !=
        "Microsoft.EntityFrameworkCore.InMemory")
        return 0;

    foreach (EntityEntry<Movie>? entry in
        ChangeTracker.Entries<Movie>()
            .Where(e => e.State == EntityState.Added))
    {
        entry.Entity.Id = _movieIdCounter++;
    }

    return base.SaveChanges();
}
```

The check on the provider is important because we don't want to override the ID creation process when we are using SQL Server.

But we are not there yet. If you run the test again you will run into another error. This time the error is that you try to update an entity that does not exist. Why?

In the Movies class, we have a piece of code that will check if the ID of the movie is 0. If so, the state is set to “Added”. If the ID is not 0, the state is set to “Modified”. The ID is set to 3 when we run our test, but it needs to be added. Let's change the Movies.CreateOrUpdate() a bit:

```
public void CreateOrUpdate(Movie movie)
{
    dataContext.Movies.Attach(movie);
    dataContext.Entry(movie).Property("LastModifiedDate").CurrentValue
    = DateTime.Now;

    if (dataContext.Entry(movie).State == EntityState.Unchanged &&
        movie.Id > 0)
    {
        dataContext.Entry(movie).State = EntityState.Modified;
    }

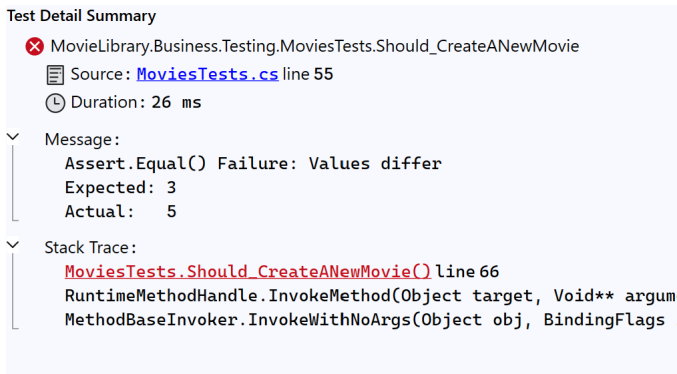
    dataContext.SaveChanges();
}
```

When the movie has a state of unchanged and the ID is greater than 0, the state should be set to “Modified”, making Entity Framework Core update the entity. The state of an attached entity is by default

“Added”.

Reset the in-memory database

But yet again: We are not there yet. Although the movie is now created you will get an error when you run both unit tests:



This too has a good explanation: The constructor is called each time xUnit runs a unit test. This will result in seeding the in-memory database multiple times because it's in memory. To fix this, you should empty the entities before seeding them:

```
...  
public MoviesTests()  
{  
    DbContextOptions<DataContext> options = new  
    DbContextOptionsBuilder<DataContext>()  
        .UseInMemoryDatabase(databaseName: "InMemoryDatabase")  
        .Options;  
    DataContext dataContext = new(options);  
  
    dataContext.Movies.RemoveRange(dataContext.Movies);  
    dataContext.Movies.AddRange(  
        new Movie()  
        {  
            ...  
        }  
    );  
}
```

With **RemoveRange()** you can remove all the movies that are placed in its parameter, in this case, all the movies. You don't have to call **SaveChanges()**, because it's committed, but not submitted yet. Entity Framework Core will remove all the rows when the **SaveChanges()** is called after the **AddRange()**.

Final words

And there you have it: xUnit with Entity Framework Core, by using an in-memory database. Pretty simple to set up.

I could show you all the tests that we can think of for the `Movies` class, but I think that is a waste of your time and mine; it's pretty straightforward and unit testing is not the subject of this book. I did, however, create more unit tests in the type-along project (Completed branch).

There are other ways of handling Entity Framework Core with unit testing. You could mock the `DataContext`, but you still need to add data on your own. You could write a fixture, but you still need to add data on your own. The way I showed you still uses the whole `DataContext`, with a minor override on the `SaveChanges()`. Also, this way you are already prepared for a scenario when you want to use an in-memory database for your application, outside of unit testing.

Another way of avoiding the whole `DataContext`, `DbContext`, `DbContextOptionsBuilder`, etc. is to use repositories. I was a big fan of these, but now I think it's redundant. Entity Framework Core is already a sort of repository, so why build another one? The main reason to use repositories is to separate the behavior of certain entities in certain situations and to make unit testing a bit easier. But Entity Framework Core is bigger and better now.

Appendix I: To SaveChanges or not to SaveChanges

The SaveChanges has been shown and mentioned a lot. You use this one when you want to insert, delete, or update data in your database. You literally save your changes to entities in the database.

A particular question I get a lot is

Why not call Entity Framework SaveChanges after each insert, update, or delete?

You could do that, but there is a theory about not doing that. In this chapter, I want to show you the differences.

To the test!

Let's take a small piece of code that you can grab from the MovieLibrary solution. In the Movies class (MovieLibrary.Business) is a method called **Create()**. I will be using this as an example.

I'll modify it by removing the check on the movie parameter and I'll remove the SaveChanges();

```
public void Create(Movie movie)
{
    dataContext.Movies.Add(movie);
}
```

In the console app, I will add a for-loop that adds 10 random movies:

```
for (int i = 0; i < 10; i++)
{
    movies.Create(new() {
        Title = $"Movie{i}Test",
        Plot = $"MoviePlot{i}_Test",
        Rating = 1,
        ReleaseDate = DateTime.Now,
        Seen = false
    });
}
```

Nothing will happen. The new movie is not added to the database. To make Entity Framework commit the changes, we need to add that SaveChanges. But there are two places where we can place them:

- After the Add in the Create method
- After (and outside) the for-loop

Both are possible, but performance-wise there are big differences. Let's start with the first option: After the Add in the Create method.

```
public void Create(Movie movie)
{
    dataContext.Movies.Add(movie);
    dataContext.SaveChanges();
}
```

Next, I run the console app and let 10 random movies be added to the database. I keep an eye on the log.

Good to know is that I set a simple logging in the DataContext and let write the log to the console-output with a minimal log level of 'info'.

This is the output:

```
Microsoft Visual Studio Debu X + -
```

```
SET IMPLICIT_TRANSACTIONS OFF;
SET NOCOUNT ON;
INSERT INTO [Movies] ([Plot], [Rating], [ReleaseDate], [Seen], [Title])
OUTPUT INSERTED.[Id]
VALUES (@p0, @p1, @p2, @p3, @p4);
info: 07/06/2024 11:20:56.562 RelationalEventId.CommandExecuted[2081] (Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (9ms) [Parameters:[@p0='?' (Size = 4000), @p1='?' (DbType = Int32), @p2='?' (DbType = DateTime2), @p3='?' (DbType = Boolean), @p4='?' (Size = 4000)], CommandText:'Text', CommandTimeout=30']
SET IMPLICIT_TRANSACTIONS OFF;
SET NOCOUNT ON;
INSERT INTO [Movies] ([Plot], [Rating], [ReleaseDate], [Seen], [Title])
OUTPUT INSERTED.[Id]
VALUES (@p0, @p1, @p2, @p3, @p4);
info: 07/06/2024 11:20:56.563 RelationalEventId.CommandExecuted[2081] (Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (9ms) [Parameters:[@p0='?' (Size = 4000), @p1='?' (DbType = Int32), @p2='?' (DbType = DateTime2), @p3='?' (DbType = Boolean), @p4='?' (Size = 4000)], CommandText:'Text', CommandTimeout=30']
SET IMPLICIT_TRANSACTIONS OFF;
SET NOCOUNT ON;
INSERT INTO [Movies] ([Plot], [Rating], [ReleaseDate], [Seen], [Title])
OUTPUT INSERTED.[Id]
VALUES (@p0, @p1, @p2, @p3, @p4);
info: 07/06/2024 11:20:56.564 RelationalEventId.CommandExecuted[2081] (Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (9ms) [Parameters:[@p0='?' (Size = 4000), @p1='?' (DbType = Int32), @p2='?' (DbType = DateTime2), @p3='?' (DbType = Boolean), @p4='?' (Size = 4000)], CommandText:'Text', CommandTimeout=30']
SET IMPLICIT_TRANSACTIONS OFF;
SET NOCOUNT ON;
INSERT INTO [Movies] ([Plot], [Rating], [ReleaseDate], [Seen], [Title])
OUTPUT INSERTED.[Id]
VALUES (@p0, @p1, @p2, @p3, @p4);
info: 07/06/2024 11:20:56.565 RelationalEventId.CommandExecuted[2081] (Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (9ms) [Parameters:[@p0='?' (Size = 4000), @p1='?' (DbType = Int32), @p2='?' (DbType = DateTime2), @p3='?' (DbType = Boolean), @p4='?' (Size = 4000)], CommandText:'Text', CommandTimeout=30']
SET IMPLICIT_TRANSACTIONS OFF;
SET NOCOUNT ON;
INSERT INTO [Movies] ([Plot], [Rating], [ReleaseDate], [Seen], [Title])
OUTPUT INSERTED.[Id]
```

It is not that easy to read, but this is a small part of the log that shows all the creations of movies, which are 10.

Let's remove the `SaveChanges` from the `Create` method and call the `SaveChanges` after the `for`-loop. To do this I have to create a method in the `Movies` class because I can't reach the `DataContext` from the console application.

```
public void Create(Movie movie)
{
    dbContext.Movies.Add(movie);
}

public void SaveChanges() {
```

```

dataContext.SaveChanges();
}

for (int i = 0; i < 10; i++)
{
    movies.Create(new() {
        Title = $"Movie{i}Test",
        Plot = $"MoviePlot{i}_Test",
        Rating = 1,
        ReleaseDate = DateTime.Now,
        Seen = false
    });
}

movies.SaveChanges();

```

Let's run it and see the result:

```

Info: 07/06/2024 11:27:03.817 RelationalEventId.CommandExecuted[20101] (Microsoft.EntityFrameworkCore.Database.Command)
      Executed DbCommand (20ms) [Parameters=[@p0="?" (Size = 4000), @p1="?" (DbType = Int32), @p2="?" (DbType = DateTime2), @p3="?" (DbType = Boolean), @p4="?" (Size = 4000), @p5="?" (Size = 4000), @p6="?" (DbType = Int32), @p7="?" (DbType = DateTime2), @p8="?" (DbType = Boolean), @p9="?" (Size = 4000), @p10="?" (DbType = Int32), @p11="?" (DbType = Boolean), @p12="?" (DbType = DateTime2), @p13="?" (DbType = Boolean), @p14="?" (Size = 4000), @p15="?" (DbType = Int32), @p16="?" (DbType = Boolean), @p17="?" (DbType = DateTime2), @p18="?" (DbType = Boolean), @p19="?" (Size = 4000), @p20="?" (DbType = Int32), @p21="?" (DbType = DateTime2), @p22="?" (DbType = Boolean), @p23="?" (DbType = Boolean), @p24="?" (Size = 4000), @p25="?" (Size = 4000), @p26="?" (DbType = Int32), @p27="?" (DbType = DateTime2), @p28="?" (DbType = Boolean), @p29="?" (Size = 4000), @p30="?" (Size = 4000), @p31="?" (DbType = Int32), @p32="?" (DbType = DateTime2), @p33="?" (DbType = Boolean), @p34="?" (Size = 4000), @p35="?" (Size = 4000), @p36="?" (DbType = Int32), @p37="?" (DbType = DateTime2), @p38="?" (DbType = Boolean), @p39="?" (Size = 4000), @p40="?" (DbType = Int32), @p41="?" (DbType = DateTime2), @p42="?" (DbType = Boolean), @p43="?" (Size = 4000), @p44="?" (Size = 4000), @p45="?" (DbType = Int32), @p46="?" (DbType = Boolean), @p47="?" (DbType = DateTime2), @p48="?" (DbType = Boolean), @p49="?" (Size = 4000)], CommandType="Text", CommandTimeout=30]
      SET IMPLICIT_TRANSACTIONS OFF;
      SET NOCOUNT ON;
      MERGE [Movies] USING (
        VALUES (@p0, @p1, @p2, @p3, @p4, 0),
        (@p5, @p6, @p7, @p8, @p9, 1),
        (@p10, @p11, @p12, @p13, @p14, 2),
        (@p15, @p16, @p17, @p18, @p19, 3),
        (@p20, @p21, @p22, @p23, @p24, 4),
        (@p25, @p26, @p27, @p28, @p29, 5),
        (@p30, @p31, @p32, @p33, @p34, 6),
        (@p35, @p36, @p37, @p38, @p39, 7),
        (@p40, @p41, @p42, @p43, @p44, 8),
        (@p45, @p46, @p47, @p48, @p49, 9)) AS i ([Plot], [Rating], [ReleaseDate], [Seen], [Title], [_Position]) ON i=0
      WHEN NOT MATCHED THEN
        INSERT ([Plot], [Rating], [ReleaseDate], [Seen], [Title])
      VALUES (i.[Plot], i.[Rating], i.[ReleaseDate], i.[Seen], i.[Title])
      OUTPUT INSERTED.[Id], i._Position;

```

Instead of 10 different queries, we now have just one, big query. This is performance-wise better and more data-efficient. If you work with a database in the cloud you might pay per transaction. You don't want to send 10 different queries; one is better.

Be careful though

Although this saves time and communication with your database, you need to be careful. If you need to connect an entity to a new entity, but both don't exist yet, you can run into problems.

Also, when you want to delete something you are better off committing that delete as soon as you can. If you don't, other requests can still use an entity that is pending for deletion. The same goes for an update.

Think carefully when you want to commit a transaction to your database. Sometimes it's better to do it right away, sometimes it's better to put more queries in the queue.

My advice: Handle one type of entity and then call the SaveChanges. This way, other entities that may rely on the previous entity have new, updated data.

All: Encrypting data

The data you store from your application through Entity Framework Core to your database is just plain information. When you are working with sensitive data it is a good idea to encrypt the information.

There are a few ways to encrypt and decrypt information from C#, but Entity Framework Core comes with a mechanism to automatically encrypt and decrypt data when you use it.

Value converter

The Value converter in Entity Framework Core is a feature that allows us to transform (or convert) data between your application and the database. When you store data in the database, the value converter can change the format or type of the data.

It also works the other way around. When we retrieve data, the value convertor can convert the information back to the original format or type your application expects.

A value converter has two parameters, both expressions. One is how to handle data from the database and the other one is how to handle data to the database. Both need to return a data type, which is defined as the generic parameters:

```
ValueConverter<string, string>(string, string)
```

This is ideal for encrypting and decrypting information from to and from the database. Let's use it!

Encrypting and decrypting

First, we create a class that handles the encrypting and decrypting. Here you can use any encryption algorithm you want; I am not going into details here.

```
public static class EncryptionHelper
{
    public static string Encrypt(string data)
    {
        // Encrypt the data
        return data;
    }
}
```

```

    }

    public static string Decrypt(string data)
    {
        // Decrypt the data
        return data;
    }
}

```

Next, we need to create a value converter in the `OnModelCreating()` method inside the `DataContext`:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    ValueConverter<string, string> converter = new(
        x => EncryptionHelper.Encrypt(x),
        x => EncryptionHelper.Decrypt(x));
    ...
}

```

We can now use this variable **converter** on a property of an entity:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    ValueConverter<string, string> converter = new(
        x => EncryptionHelper.Encrypt(x),
        x => EncryptionHelper.Decrypt(x));

    modelBuilder.Entity<Movie>()
        .Property(x => x.Title)
        .HasConversion(converter);
    ...
}

```

When used, the data of `Title` will be encrypted when sent to the database and decrypted when retrieved from the database.

Some cons

Apart from the fact you can use any encryption algorithm like this and it all sounds pretty convenient, there are some serious cons.

Encrypting data like this limits the search capabilities in the database tables. Because you can't query the information like we have been doing with the `WHERE`, you can't use it like this when the data is encrypted.

You will need more storage if you want to encrypt the data. This is not only when you use a value converter, but also with 'normal' encryption.

Encrypting and decrypting will cause a decrease in performance. You

should consider how you are handling encryption without slowing down the application.

Conclusion

And that concludes our adventure into Entity Framework Core. We have been working from SQL to a fully working API and console app with an implementation of Entity Framework Core.

We have looked at so many subjects that make Entity Framework Core work. First, we looked at how we can refactor an application from using hardcoded data to using data from a database through Entity Framework Core. A glance at **entities**, the **DbContext**, the **connection string**, and the **DbSet**. But we also looked at **migrations** a bit.

But then the real work (learning) started. We have learned what entities are and how we can set them up. A bit of a short chapter, but this subject came back multiple times throughout this book. We implemented the **MaxLength** and looked at the **Key**, **Column**, and **Required** attributes. We have learned about **DatabaseGenerated** and how we can have properties inside an entity that will not be created in the database table using **NotMapped**.

From that, we moved to **managing migrations**. Migrations are just as important as entities because they are the way to manage the structure and some of the data inside your database. We already learned how we can make migrations in the previous chapter, but now we looked into other actions. Things like **reverting migrations**. Because you can't simply delete the migration file, but you need to undo the migration in the database, then delete it in a way you also update the snapshot.

But adding and removing migrations isn't everything you can do. You can also **add your own SQL** to a migration, add stored produce, and use them for seeding the database.

I also added some extras to this chapter. Like **scripting the migrations** to an SQL file, starting over by **dropping the database**, and **listing the migrations** to get an overview of the status.

Relationships between entities are something you will need. Just like the JOINS in SQL, we need to combine information between different entities to get a complete picture. That's why we also looked at **conventions** and **navigations**; the way Entity Framework Core connects properties of entities.

Next up was the **one-to-one relationship**, where one entity has one relationship with another entity. In our example, this was the Movie and Genre; each movie has one genre. This subject also brought along the related data query, where he learned about the 3 types of loading the related data: **Lazy**, **implicit**, and **eager**.

But there are also situations in which one entity has a relationship with more than one other entity. This is called a **one-to-many relationship**. This one brings along the **ICollection** and the **composite key**.

Stored procedures are a different subject. They are not entities, so we can't create them in the same way as an entity. We need a migration to execute SQL on the database to create the stored procedure. Then we can connect the result of the stored procedure to an entity or not. If not, we need to **remove** the need to use a **primary key** by Entity Framework Core.

We also learned that when the result of the stored procedure is connected to an entity, Entity Framework Core **tracks that data** and we can update it if needed.

Time to implement the API. We have stored the connection string inside the appropriate **appsettings** and retrieved it in the Program.cs. We then used the **AddDbContext** to set up the DataContext, which inherits the DbContext.

We can now inject the DataContext through **dependency injection** throughout our application.

This does mean the way we manage migrations changes. We need to have the **API** set as a **startup project** and the **MovieLibrary.Business** is set as the default **project** in the **Package Manager Console**. To create migrations and update the database through the API, we need to install another package.

Since the whole structure of the DataContext has changed, we also fixed the console application.

Logging is an important part of getting information from our application. That is why we implemented the simple way of logging with Entity Framework Core. Simply logging information in the **console** or **output window**. We have to be careful, but we can also log **sensitive data** since this is turned off by default. Then we looked at **log levels** and how we can use them.

But there is another way to log information: **Microsoft logging**. We created a **LoggerFactory**, which can also be used in other parts of the application.

Another form of logging is **query interceptors**. They intercept a query before it's sent to the database or catch the information that is returned. We can change the **query** before it's sent or **modify** the returned **information** before the application uses it. We can also use the query interceptors to **monitor** the **performance**.

Security is always an issue and the query interceptors can help with this. They can determine if a query request is allowed or not.

Entities are **tracked** by Entity Framework Core, meaning that are actively connected to the data in the database through Entity Framework Core. This makes **updating** the data way easier. But if an object is an entity, but it's not tracked we can **attach** it to Entity Framework.

But tracking does cost some **performance**. Not much, but if you have an active (web) application it might show. That's why we can add **no tracking** to the retrieval of the entities. But this will make us lose the **identity resolution**. Luckily, Entity Framework Core can add it again without tracking the entities.

Lastly, we look at how **stored procedures** are tracked, or not.

Although we already used it, we learned more about **Fluent API**. It's a way to set up entities and the database through C# code and chaining. We set up the entities the same way we did earlier, but now with Fluent API.

But Fluent API can also do things we can't do with attributes, like **shadow properties**. These are the opposite of the NotMapped: The field is in the database table, but not in the entity.

Finally, we learned how we can **unit test** with Entity Framework Core by using an **in-memory database**. This chapter is not only for unit testing but also for how you can use a different database provider with Entity Framework Core.

The appendixes are an extra. They are a more to-the-point description of certain subjects.

And that's Entity Framework Core for you. Know that Entity Framework Core is much bigger than what you have learned here. But

this book gives you a headstart on the possibilities that this framework gives you.

I hope you have learned a lot and enjoyed this 'little' e-book... I know I did enjoy creating it!



This book is for those who want to learn how to use Entity Framework from zero or those who want to learn a little bit extra about this subject. Don't expect 100% coverage on the whole Entity Framework Core subject, because that's impossible. I will give you enough information to work with in a way you can implement Entity Framework in your own application(s).

Most books I have read are so focused on theory and working with the perfect scenario. In my opinion, this isn't right. There isn't a perfect scenario in a perfect environment.

That is why I place my focus more on the practical side of things. I want to show you how it is done, rather than just tell you. This way I will tell you less unused theory, but more practical examples.

To help you with the practical side, I have included examples and a project which you can download from my GitHub. With this project, you can type along as we are building more aspects of Entity Framework Core in a console application and API.

You do need to know the basics of C#! Know what variables are, how to create classes and methods, how dependency injection works, how to make decisions, how unit testing works, etc.

Website:

<https://kenslearningcurve.com>



KENS LEARNING CURVE
LOVE TO LEARN